Dandelion Microcode Reference

5 Dec 80
R. Garner

This document describes revisions G-L of the Dandelion CP,
MASS, and the CP Panel of Burdock.
(previous revisions: 31 Oct 79, 8 Jan 80, 22 Jan 80, 14 Feb 80)

XEROX

Contents

Where the files are saved:

[Iris]<Workstation>Mass>MASS.bcd   {Mesa 6.0}
[Iris]<Workstation>Jarvis>Burdock60.dm   {Mesa 6.0}
     and Burdock60.cm
[Iris]<Workstation>mc>Kernel.fb

[Iris]<Workstation>LH>DMR.press
     and DMR.dm

A. Introduction

This manual describes the Dandelion central processor, the CP, from a microprogramming perspective. This user's manual includes example microcode statements, application notes, and pointers to restrictions which the programmer should eventually confront.

You should become thoroughly familiar with figures 1 and 2, the microinstruction format and the data paths. When in doubt about the construction of a particular microinstruction statement, it is only necessary to submit it to the assembler (MASS). However, MASS may reject a perfectly reasonable statement due to incorrect register assignments (sec. E.3) or timing violations which may be permissible with 4 or 8 bit wide arguments (sec. Q).

The complete Dandelion system is described in the Dandelion Hardware Manual (not available at this time). In general, the CP is shared among the IO devices in a fixed round-robin fashion. Each device has an assigned *click* during which its *task* microcode can run if the hardware asserts a wakeup request. The five clicks make up a *round*. If a device does not utilize its click, the Mesa Emulator executes during the click instead. In general, microinstructions can be specified without regard to click boundaries since the hardware will save the micro-program counters and branch conditions between *task* changes. There are no task specific registers, i.e., all registers can be addressed by all tasks.

Click	Task
0	Ethernet
1	Disk (SA4000/SA1000)
2	IOP (i8085 low speed IO system)
3	Ethernet
4	Display/LSEP

Three microinstructions are executed per click (notationally, c1, c2, and c3), while one word can be moved to and/or from the memory in this time. A cycle is 137 nanoseconds long, making a click 411 nSec and a round about 2 microseconds.

Figure 1 shows the microinstruction format. The first word generally controls the 2901 based ALU: *aS* selects the two ALU operands (notationally, *R,S*), *aF* determines the function to be applied between them, and *aD* (along with *sh*, a decoding of the *fX* and *fY* fields) specifies the ALU destination of the output. Either the ALU output or the R register given by the *rA* field (called A-bypass) is placed on the Y bus. Note that the following values of *aS* imply use of an X bus operand: *D,A, D,Q,* and *D,0.*

fX, fY, and *fZ* are multipurpose function fields. The interpretation of *fY* and *fZ* is determined by the value of *fS*. *fS[0-1]* describes how *fY* is decoded: *fYNorm, DispBr* (dispatch/branch), *IOOut,* or the high 4 bits of a byte constant (*Byte*). *fS[2-3]* describes how *fZ* is decoded: *fZNorm,* low 4 bits of a U register address, *IOXIn,* a 4 bit constant (*Nibble*), or the low 4 bits of a byte constant. *fS[2-3]* also determines SU register addressing (sec. E).

Figure 2 shows the CP data paths: the two major 16-bit buses, X and Y, are identified and the 2901 ALU is enclosed within the dashed rectangle. The box labeled LRotn is the multiplexer which can rotate the Y bus by 0, 4, 8, or 12 bit positions (sec. C.2). Although there are many possible register-to-register operations implied by this figure, the complete set is not available due to the microinstruction encoding scheme and timing considerations.

rA	rB	aS	aF	aD	ep	Cin	enSU	mem	fS	fX	fY	fZ	INIA

0   4   8   11   14   16   20   24   28   32   36   47
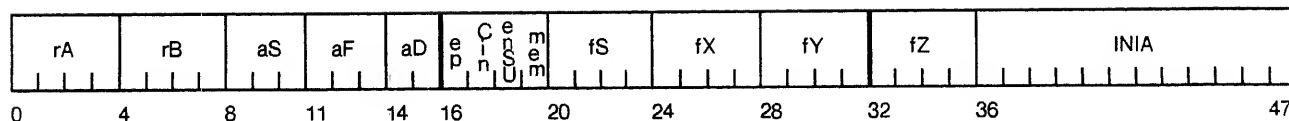
Field	Description
rA	2901 A reg addr, U addr [0-3]
rB	2901 B reg addr, RH addr
aS	2901 alu Source operand pair
aF	2901 alu Function
aD	2901 alu Destination/shift control
ep	Even Parity
Cin	2901 Carry In, Shift Ends, writeSU (if enSU = 1)
enSU	enable SU reg file
mem	MAR← (if c1), MDR← (if c2), ←MD (if c3)
fS	Function field Selector
fX	X Function
fY	Y Function
fZ	Z Function
INIA	Next Instruction Address

aS	R,S
0	A, Q
1	A, B
2	0, Q
3	0, B
4	0, A
5	D, A
6	D, Q
7	D, 0

aF	F
0	R + S
1	S − R
2	R − S
3	R or S
4	R and S
5	~R and S
6	R xor S
7	~R xor S

sh,,aD	R[rB]←	Q←	Ybus←
0	no write	F	F
1	no write	no write	F
2	F	no write	A
3	F	no write	F
4	F/2	Q/2	F
5	F/2	no write	F
6	2F	2Q	F
7	2F	no write	F

sh ← (fX = shift) OR (fX = cycle) OR (fY = cycle)

fS[0-1]	fY←
0	DispBr
1	fYNorm
2	IOOut
3	Byte

fS[2-3]	fZ←	SU addr[0-7]			
0	fZNorm	0,,stackP			
1	Nibble	0,,stackP			
2	Uaddr[4-7]	rA,,fZ	rA,,Y[12-15]*	IF	fZ = AltUaddr*
3	IOXIn	rA,,fZ	rA,,Y[12-15]*	IF	fZ = AltUaddr*

* as executed by previous u-instr

fX	fXNorm	fY	fYNorm	DispBr	IOOut	fZ	fZNorm	IOXIn
0	pCall/Ret0	0	ExitKern	NegBr	IOPOData←	0	Refresh	←EIData
1	pCall/Ret1	1	EnterKern	ZeroBr	IOPCtl←	1	IBPtr←1	←EStatus
2	pCall/Ret2	2	ClrIntErr	NZeroBr	KOData←	2	IBPtr←0	←KIData
3	pCall/Ret3	3	IBDisp	MesaIntBr	KCtl←	3	Cin←pc16	←KStatus
4	pCall/Ret4	4	MesaIntRq	PgCarryBr	EOData←	4	BankR←	KStrobe
5	pCall/Ret5	5	stackP←	CarryBr	EICtl←	5	pop	←MStatus
6	pCall/Ret6	6	IB←	XRefBr	DCtlFifo←	6	push	←KTest
7	pCall/Ret7	7	cycle	NibCarryBr	DCtl←	7	AltUaddr	EStrobe
8	Noop	8	Noop	XDisp	DBorder←	8	Noop	←IOPIData
9	RH←	9	Map←	YDisp	PCtl←	9	Noop	←IOPStatus
A	shift	A	Refresh	XC2npcDisp	MCtl←	A	Noop	←ErrnIBnStkp
B	cycle	B	push	YIODisp		B	Noop	←RH
C	Cin←pc16	C	ClrDPRq	XwdDisp	EOCtl←	C	LRot0	←ibNA
D	Map←	D	ClrIOPRq	XHDisp	KCmd←	D	LRot12	←ib
E	pop	E	ClrRefRq	XLDisp		E	LRot8	←ibLow
F	push	F	ClrKFlags	PgCrOvDisp	POData←	F	LRot4	←ibHigh

Figure 1. Dandelion Microinstruction Format

3 Sept 80

4×2901 ALU

pc16, 0, 1

~Cin

~r.15

Cin,r.0,q.0

Cin,r.15,Carry

R registers

Q

B
A

0

Q
0
B
A
D

Cin
S
F
R

F bus

A-bypass

Y bus

14 pa.

16

16

Y bus

LRotn

X bus

SU
registers

stackP

4

4

RH
registers

8

8

YH bus

8

MAR

16

MDR

16

Main
Memory

22-bit VA
18-bit RA

MD

16

Instruction
Buffer

16

ibHigh

ib

ibLow

4

8

4

fZ   (Nibble constant)

fY,,fZ   (Byte constant)   4

IOIn   8

IOOut

IOOut

Y-bus branch:
    YDisp

ALU branches:
    ZeroBr
    NZeroBr
    NegBr
    NibCarryBr
    PgCarryBr
    CarryBr
    PgCrOvDisp

X-bus branches:
    XHDisp   X.4,,X.0
    XLDisp   X.8,,X.15
    XwdDisp   X.9,,X.10
    XRefBr   X.11
    XDisp   X[12-15]

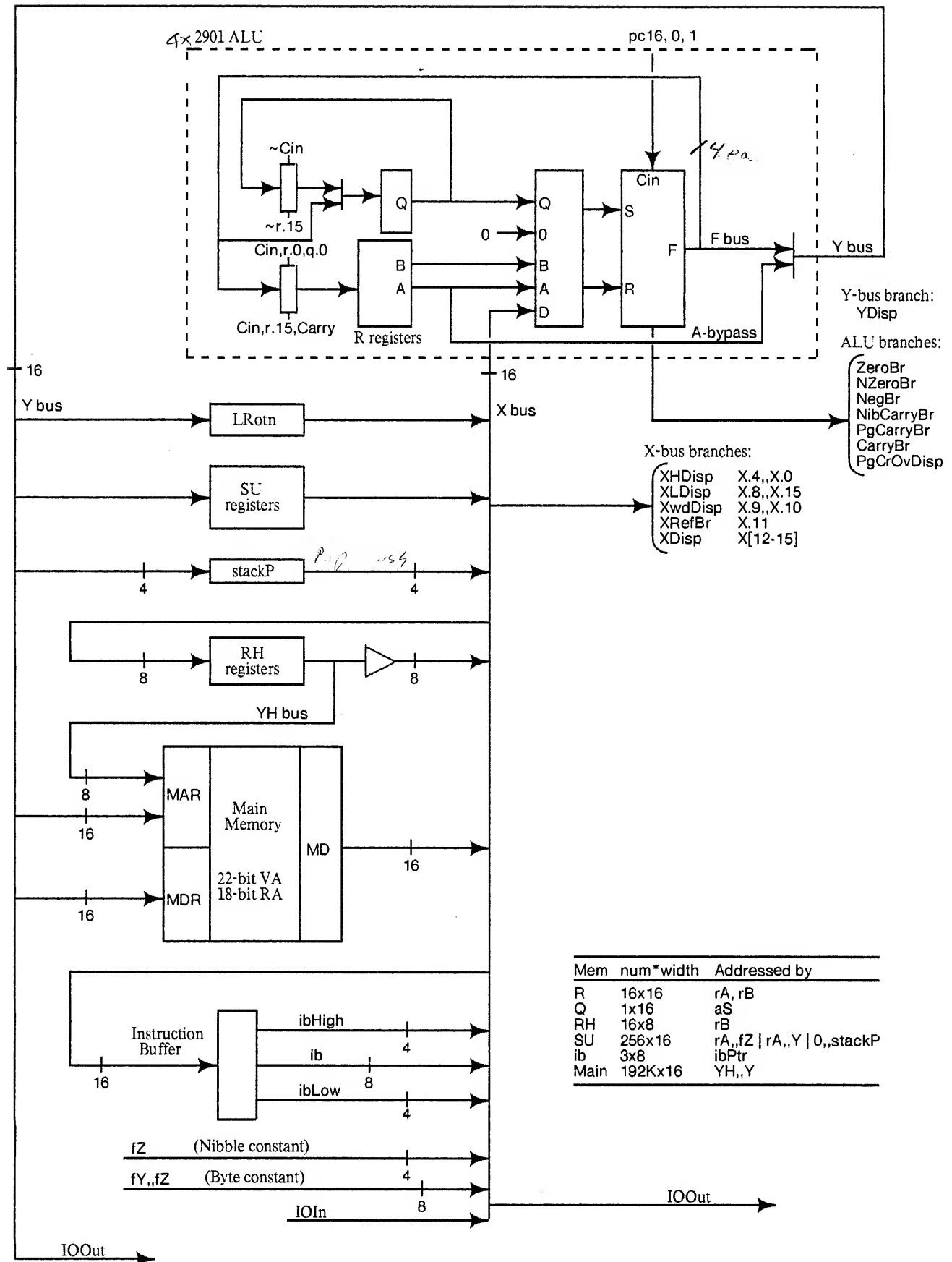| Mem	num*width	Addressed by
R	16x16	rA, rB
Q	1x16	aS
RH	16x8	rB
SU	256x16	rA,,fZ | rA,,Y | 0,,stackP
ib	3x8	ibPtr
Main	192Kx16	YH,,Y

Figure 2. Dandelion Processor Data Paths

A. Registers & Constants

There are 5 register sets available to the microprogrammer: The 16 R registers inside the ALU (2901), the corresponding 16 RH registers on the X bus, the 256 "slow" SU registers on the X bus (sec. E), the 8 Link registers (sec. D), and the 3 byte instruction buffer (sec. G). Twenty-four input/ouput registers (called *IOXIn* or *IOOut*) on the X and Y buses are also directly addressable via the *fY* or *fZ* microinstruction fields.

Also available are 5 registers which are reserved by the Emulator in the standard microcode system: The Q register, *stackP* (sec. F), *ibPtr* (sec. G), *pc16* (sec. G.10), and *MInt* (sec. M.1).

1. R Registers: The 16-bit wide R register memory has two ports: the selected word (notationally, A) from the A-port is addressed by the *rA* field and from the B-port (B) by the *rB* field. If *rA* equals *rB*, the same R register appears at both ports. Thus, any two R registers (or a single register specified via *rA* and/or *rB*) can be operands of an ALU computation.

An R register is written from the ALU output F bus as specified by the *rB* microinstruction field. The ALU output can be shifted left (notationally, **2F**) or right (**F/2**) one bit position before being written.

In the standard system, 7 registers are allocated to the Emulator (TOS, L, G, PC, and 3 temporaries), 3 to the Display/LSEP, and 2 each for the Disk, Ethernet, and IOP.

2. RH Registers: The 8 bit wide RH registers are read and written from the X bus. They normally act as extensions of the R register file and hold high-order memory address bits (sec. J & K). However, they can be used as flags, subroutine return points (sec. D.6), and general byte storage. An RH register is always addressed by the *rB* field of the microinstruction (which also specifies the R register write address).

3. Q Register: The 16-bit wide Q register is addressed by the *aS* field and is loaded from the ALU output which can be pre-shifted left or right 1 bit. Note that the Q register can not be simultaneously written along with an R register or be loaded in an instruction which uses A-bypass ($aD = 2$). The Q register can be used in double length shifts (sec. C).

4. Constants: Four-or-eight-bit constants can be placed onto the X bus where they can be used in branching, be loaded into X bus destination registers, or be an ALU operand. Four-bit constants (Nibble) use the *fZ* field and 8 bit constants (Byte) use the *fY* and *fZ* fields. The upper 12 or 8 bits, respectively, are zeroed.

Larger constants can be preloaded into U registers and used like normal constants (except for timing, sec. Q). Zero is available inside the ALU and does not utilize the X bus. ALU " + 1" or "-1" operations are also possible without the X bus since they an artifact of *Cin*. Without preloading another register, U registers can only be written in one statement with 0 or -1.

```
Reg ← -0E,   {Reg ← 0FFF2}                        c1;
Reg ← ~0FF,   {Reg ← 0FF00}                        c2;
Reg ← ~Reg xor Reg,   {Reg ← 0FFFF}               c3;

Ureg ← 0,                                          c1;
Ureg ← ~Reg xor Reg,   {Ureg ← -1}                 c2;
```

Sixteen-bit constants with identical halves can be constructed in two cycles instead of the three normally required in the general case.

```
R ← 0AA,                                           c1;
R ← (R LRot8) or R,   {R ← 0AAAA}                   c2;
```

B. Branches & Dispatches

The following table lists the source for each branch/dispatch function and where they are OR'd into *INIA*. See section D on the Link register dispatches and section G for the IB dispatch.

Some notation: The address of an instruction is called *IA*. *NIA*, Next Instruction Address, is the 12-bit quantity which addresses the control store, and is the address of the instruction to be executed in the next cycle. *INIA* refers to the contents of the 12-bit microinstruction field. In the hardware, *INIA* is OR'd with the currently specified dispatch/branch bits to form *NIA*.

	source	INIA dest	
NegBr	F.0	11	sign of alu result (This is not Y.0)
ZeroBr	F = 0	11	alu output equal to zero
NZeroBr	F # 0	11	alu output not equal to zero
MesaIntBr	MesaInt	11	Mesa Interrupt bit
NibCarryBr	Cout.12	11	alu carry out of low Nibble
PgCarryBr	Cout.8	11	alu carry out of low Byte
CarryBr	Cout.0	11	alu carry out
XRefBr	X.11	11	present & referenced map bit
XwdDisp	X.9,,X.10	10,,11	X bus write protect & dirty bits
XHDisp	X.4,,X.0	10,,11	X High bus
XLDisp	X.8,,X.15	10,,11	X Low bus
PgCrOvDisp	PgCross,,OVR	10,,11	pageCross & Overflow bits
XDisp	X[12-15]	8,,9,,10,,11	low 4 bits of Xbus
YDisp	Y[12-15]	8,,9,,10,,11	low 4 bits of Ybus
XC2npcDisp	X[12-13],,C2,,~pc16	8,,9,,10,,11	X bus, cycle2, ~pc16
YIODisp	Y[12-13],,bp.39,,bp.139	8,,9,,10,,11	IO branches (bp = Backplane pin)
IBDisp	ib	[4-11]	Instruction Buffer
LnDisp	Linkn	8,,9,,10,,11	Linkn dispatch (n = 0..7)

1. Simple Branches: Branches require 2 cycles to complete. In the first microinstruction the branch or dispatch condition is specified by a value in the *fY* field of the microinstruction. During execution of the first instruction, the hardware OR's the result {0,1} of the specified branch into *INIA[11]* of the second instruction, which is being read from the control store. This will send control to either *INIA* or *INIA OR 1*.

In the source program, the second instruction should contain a BRANCH[Label0, Label1] phrase. MASS will assign **Label0** to an even control store location and **Label1** to (**Label0** OR 1). Note that the hardware treats the BRANCH macro like a GOTO macro, i.e., the branch condition specified in a previous microinstruction is always OR'd into the *INIA* field of the current instruction.

```
        Reg ← Reg xor RegA, ZeroBr,                                c1;
        BRANCH[NotZero, Zero],                                     c2;

NotZero:    Noop   {here if Reg # RegA},                           c3;
Zero:       Noop   {here if Reg = RegA},                           c3;
```

2. Address Constraints: The at[x, y, Label] macro is used to constrain the control store location of instructions. It causes MASS to place the instruction at an address which is (x MOD y) and in the same "MOD group" as the instruction at **Label**. The third argument (**Label**) is optional.

Although the at's are not required, the above example could be rewritten as:

```
        Reg ← Reg xor RegA, ZeroBr,                                c1;
        BRANCH[NotZero, Zero],                                     c2;

NotZero:    Noop,                                       c3, at[0,2,Zero];
Zero:       Noop,                                       c3, at[1,2,NotZero];
```

3. Simple Dispatches: Dispatches, like branches, require 2 cycles to complete. Dispatches differ from branches only in that they OR more than a single bit into *INIA*.

The second statement of a dispatch contains either DISP2[Label], DISP3[Label], or DISP4[Label] (abbreviated DISPn, where n specifies the number of bits used in the dispatch). When constructing the *INIA* field, MASS zeros the low n bits of Label unless there is a mask specified (see below). at clauses ARE required for all dispatches.

```
          Xbus ← MD, XLDisp,                          c3;   (MD)
          DISP2[Table],                               c2;

Table:    Noop,  {here if MD.8,,MD.15 = 0}            c3, at[0,4,Table];
          Noop,  {here if MD.8,,MD.15 = 1}            c3, at[1,4,Table];
          Noop,  {here if MD.8,,MD.15 = 2}            c3, at[2,4,Table];
          Noop,  {here if MD.8,,MD.15 = 3}            c3, at[3,4,Table];
```

4. Branching via a Dispatch: A two-way branch on what is normally a multi-way dispatch is accomplished by specifying a mask which has 1's in those bit positions of the dispatch which should be ignored by the hardware. The hardware still OR's the specified dispatch bits into *INIA*, but since the *INIA* bits given by the mask are 1, the OR operation has no effect in these positions.

The mask should not be wider than the width of the specified dispatch. The mask is a third argument to the BRANCH macro. The only legitimate values for the third argument have exactly one zero in their binary representation (and a leading zero is used if needed); they are {1,2,3,5,6,7,0B,0D,0E}. OF is illegal since it has no zero in its binary. at clauses are NOT required.

```
          Xbus ← Reg LRot8, XDisp,                                        c1;
          BRANCH[NotSet, Set, 0B],  {branch on bit 13 of X bus}           c2;

NotSet:   Noop,  {here if bit 5 of Reg = 0},                              c3;
Set:      Noop,  {here if bit 5 of Reg = 1},                              c3;
```

5. Sub-Field Dispatching: Similarly, a dispatch on a sub-field of a dispatch is specified with a mask which indicates which bits of the larger dispatch should be ignored. The mask is a second argument to the DISPn macro. at clauses ARE required. Note that the addresses of the statements in the dispatch table must also have the same bits set to 1.

```
          Xbus ← rhReg, XDisp,                             c1;
          DISP4[Table, 9],                                 c2;

Table:    Noop,  {here if RHReg.13,,RHReg.14 = 0},         c3, at[9,10,Table];
          Noop,  {here if RHReg.13,,RHReg.14 = 1},         c3, at[0B,10,Table];
          Noop,  {here if RHReg.13,,RHReg.14 = 2},         c3, at[0D,10,Table];
          Noop,  {here if RHReg.13,,RHReg.14 = 3},         c3, at[0F,10,Table];
```

6. Canceling Branches: The CANCELBR macro is used to cancel pending branch/dispatch conditions by forcing the argument address to have ones where condition bits would normally be OR'd in.

CANCELBR may be necessary after a sequence of two instructions which specify branching or may be required after a MAR← (see sec. I). MASS will give an error message where it thinks there should be a CANCELBR. (It uses the principle that all dispatches or branches or LnDisp's should be followed by either a BRANCH, DISPn, RET, or CANCELBR.)

```
          R ← R + 2, ZeroBr,                             c1;
          [] ← T - 1, NegBr, BRANCH[NZ, Z],             c2;

NZ:       BRANCH[Pos, Neg],                              c3;
Z:        CANCELBR[Zero],                                c3;

Zero:     Noop,    {placed at[1,2] by MASS}             c1;
```

9

7. Canceling Dispatches: The CANCELBR macro can also specify a mask which declares which bits of an address Label should be set to 1. The mask is a second argument to the CANCELBR macro and can have values [0..0F]. Thus, a mask of 0F causes the instruction at the argument's address to be placed at[0F,10].

CANCELBR[Label, 2] is used to cancel the implied *pageCross* branch of a MAR← (see section I).

```
        ZeroBr,                                              c1;
        pRet0, BRANCH[NZ, Z],                                c2;

NZ:     RET[NZReturn],                                       c3;
Z:      CANCELBR[NotYet, 0F],                                c3;

NotYet: Noop,    {placed at[0F,10] by MASS}                  c1;
```

8. CANCELBR masks: A general rule for the branch/dispatch masks described above: The mask always indicates which bits should be IGNORED. Also note that the low 4 bits of the addresses composing a dispatch table must have the same bits set as the corresponding mask does.

All dispatch bits need not be masked: If it is known that some bits of the specified dispatch are always zero, then those bit postions need not appear in the mask or at's of the dispatch table.

If it is known that some resulting values of a dispatch (or sub-field of a dispatch) will never occur. it is not necessary to specify these entries.

9. pageCross Branch: The *pageCross* branch OR's into $INIA[10]$ (instead of $INIA[11]$ as do other branches). A *pageCross* branch occurs with the PgCrOvDisp dispatch or automatically with a MAR←.

pageCross equals *pageCarry XOR aF.2.*   See sec. I on Memory for details.

10. Carry Branches: Even in the absence of ALU arithmetic, the NibCarryBr, PgCarryBr, and *pageCross* branches can produce non-zero results (i.e., branch). When $aF = RandS$, NibCarryBr and PgCarryBr are the logical inner product of R with S (of the low 4 or 8 bits). If $aF = notRandS$ and $aS = 0,B$, $0,A$ or $0,Q$, then NibCarryBr tests for the low nibble not equaling zero and PgCarryBr tests for the low byte not equaling zero. If $aF = RorS$, NibCarryBr is the logical inner sum of R with S.

11. Equivalent Branches: XDirtyDisp is equivalent to XLDisp and is used to dispatch on the Dirty bit of a Map entry (see sec. J).

EtherDisp is equivalent to YIODisp.   When the Option card is present, $BP[39] = 0$ and $BP[139] = Ethernet\ Attention.$ When the Option card is not plugged in, $BP[39] = 1.$

12. GOTOABS: The GOTOABS macro sends control to an absolute control store location.

```
        GOTOABS[0],                                          c3;
```

C. Shifting & Rotating

The single bit shifts and rotates (LShift1, RShift1, LRot1, RRot1) are applied to the output of the ALU (the internal F bus) and the results can only go to an R register, or the Q register on double length shifts.

The four bit rotates (LRot0, LRot4, LRot8, LRot12) are done while moving something from the Y bus to the X bus. If the result of the 4 bit rotate is destined for an R register, it must have been placed onto the Y bus via the A-bypass (which implies that $aD=2$). The 4 bit rotates are abbreviated LRotn.

Single bit shifts use the fX field, single bit rotates fX or fY, and 4 bit rotates fZ.

LShift1, RShift1	Left, Right Shift R by 1
LRot1, RRot1	Left, Right Rotate R by 1
DALShift1, DARShift1	Double Arithmetic Left, Right Shift R,,Q by 1
DLShift1, DRShift1	Double Left, Right Shift R,,Q by 1
LRot0, LRot4, LRot8, LRot12	Left Rotate of Ybus

Name	aD.1	fX, fY
LShift1, RShift1	1	shift
LRot1, RRot1	1	cycle
DALShift1, DARShift1	0	shift
DLShift1, DRShift1	0	cycle

Cin → [R] ← Cin

[R]

Cout → [R] ← ~ → [Q] ← ~Cin

Cin → [R] ← ~ → [Q] ← ~Cin

aD.0 = 0 implies right shift
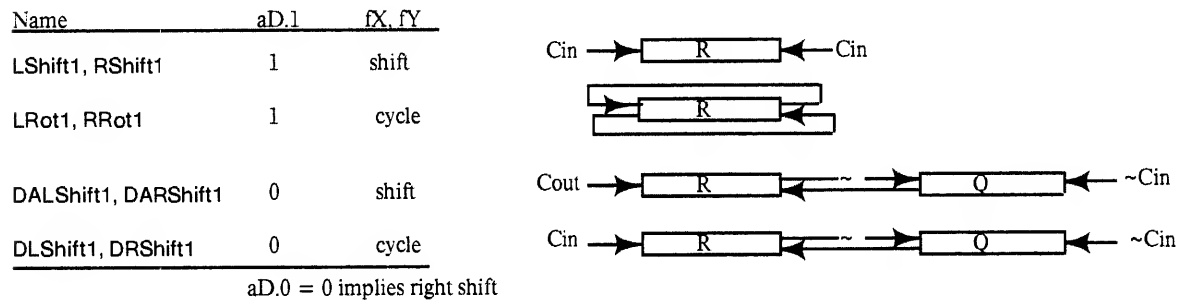
Figure 3. Single Bit Shifting & Rotating

1. Single Bit Shifts & Rotates: The single bit shift and rotate macros must precede the arithmetic clause to be shifted. Parentheses are ignored by MASS, but are included here to clarify the actual operations performed.

```
Reg ← LShift1 Reg,                                     c1;
Reg ← LShift1 (Reg and 0F)                             c2;
Reg ← LShift1 (Reg + 1)                                c3;
```

2. Four Bit Rotates: The 4 bit rotates are handled differently by MASS. When the quantity to be rotated is the output of the ALU, the rotate clause must follow the arithmetic clause. When the quantity to be rotated is a register whose rotated value will be used as an input to the ALU, the rotate phrase must follow the register to be rotated (which must be the first part of the arithmetic clause). Parentheses are ignored by MASS, but can be used to clarify the actual operations performed.

```
Reg ← Reg LRot4,                                       c1;
Reg ← (Reg LRot4) or Reg                               c2;
rhReg ← (RegA + Reg) LRot4                             c3;
```

3. Shift Ends: The macros SE←0, SE←1, or SE←pc16 are used to specify the shift ends. SE← is equivalent to Cin←. The Q register gets the INVERSE of Cin or R.0 on double length shifts.

```
Reg ← LShift1 Reg, SE←1, {puts 1 into Reg.15}          c1;
Reg ← DALShift1 Reg, SE←1, {puts 0 into Q.15}          c2;
```

DARShift1 shifts *carryOut* of the current ALU operation into the left side of the double length R,,Q. (This feature can be used by multiply routines.) Since *carryOut* can be true on logical

operations, if you want to shift a 0 into the left side you must specify an arithmetic operation which produces no carry:

```
Reg ← DARShift1 (Reg + 0),                                           c1;
```

4. Shifting and SU: Both the single bit shifts and the SU registers utilize Cin. Therefore, if an SU register is being read and shifted, or is being read and there is a shift operation in the ALU, the shift ends must be 0. If an SU register is being writen, the shift ends must be 1. Note that A-bypass and shifting is not a legal aD combination.

```
R ← Lshift1 Ureg, {SE←0,}                                           c1;
R ← Lshift1 R, Xbus ← Ureg, XDisp, {SE←0,}                          c2;
Ureg ← T, T ← T LShift1, {SE←1,}                                    c3;
```

5. Shifting and Arithmetic: Both the single bit shifts and ALU arithmetic utilize Cin. If shifting is combined with ALU addition, the shift ends must be 0 unless a + 1 operation is desired. If shifting is combined with subtraction, the shift ends must be 1 unless a -1 operation is desired. Note that Reg-Reg implies $Cin = 1$.

```
Reg ← RShift1 (Reg + Reg), {SE←0,}                                  c1;
Reg ← RShift1 (Reg + 1), {SE←1,}                                    c2;
Reg ← RShift1 (Reg - Reg - 1), {SE←0,}                              c3;
Reg ← RShift1 (Reg - Reg), {SE←1,}                                  c1;
```

6. LRotn and A-Bypass: LRotn, when used in conjunction with A-bypass, allows the ALU to be used for other purposes. For instance, an R register can be rotated and placed onto the X bus (where it can be branched on or sent to RH or IOOut) while arithmetic is performed in the ALU. Note that the R register given by rB must always be written when A-bypass is used.

```
IOOut ← RegA LRot8, Reg ← Reg + 1,                                  c1;
rhReg ← Reg LRot12, Reg ← ~Reg, XDisp,                             c2;
STK ← RegA, rhReg ← RegA LRot0, Reg ← Reg + 1,                      c3;
```

7. General 16 Bit Rotate: An arbitrary 16 bit rotate requires 3 cycles to complete (plus 1 to specify the rotation). This example uses 2 R registers and assumes the shift count is in rhReg.

```
       [] ←rhReg, XDisp,                                           c1;
       T ← LRot1 R, DISP4[Rot],                                    c2;

Rot:   GOTO[Shift0],                                               c3, at[0,10,Rot];
       R ← T, GOTO[Shift0],                                        c3, at[1,10,Rot];
       R ← LRot1 T, GOTO[Shift0],                                  c3, at[2,10,Rot];
       R ← RRot1 R, GOTO[Shift4],                                  c3, at[3,10,Rot];

       GOTO[Shift4],                                               c3, at[4,10,Rot];
       R ← LRot1 R, GOTO[Shift4],                                  c3, at[5,10,Rot];
       R ← LRot1 T, GOTO[Shift4],                                  c3, at[6,10,Rot];
       R ← RRot1 R, GOTO[Shift8],                                  c3, at[7,10,Rot];

       GOTO[Shift8],                                               c3, at[8,10,Rot];
       R ← LRot1 R, GOTO[Shift8],                                  c3, at[9,10,Rot];
       R ← LRot1 T, GOTO[Shift8],                                  c3, at[0A,10,Rot];
       R ← RRot1 R, GOTO[Shift12],                                 c3, at[0B,10,Rot];

       GOTO[Shift12],                                              c3, at[0C,10,Rot];
       R ← LRot1 R, GOTO[Shift12],                                 c3, at[0D,10,Rot];
       R ← LRot1 T, GOTO[Shift12],                                 c3, at[0E,10,Rot];
       R ← RRot1 R, GOTO[Shift0],                                  c3, at[0F,10,Rot];

Shift0:    GOTO[Done],                                             c1;
Shift4:    R ← R LRot4, GOTO[Done],                                c1;
Shift8:    R ← R LRot8, GOTO[Done],                                c1;
Shift12:   R ← R LRot12, GOTO[Done],                               c1;
```

D. Link Registers & Subroutines

Link registers, besides being used for subroutines, can be used to store 4 bits of state information which can be dispatched on later. Also, constants or branch condition bits can be stored in Link registers. Ln← (pCalln) is used to load a link register and LnDisp (pRetn) has the effect of a dispatch. When either an LnDisp or Ln← is specified, the following instruction must be constrained in some way.

The general approach to subroutines is that callers identify themselves with some unique (usually 4-bit) number which the subroutine then dispatches on to return to the caller. All subroutine call (and return) points usually have the same cycle number, but this is not necessary if the subroutine has no MAR←'s or Map←'s or any other cycle-dependent macros.

In the standard microcode system the Mesa Emulator uses the first 4 link registers, while the remaining are utilized one per device task.

In general, Link registers are loaded from the low 4 bits of *NIA*. This value is a constant if there are no branch/dispatch bits pending from the previously-executed instruction.

1. Loading Link with Constant: The format for loading a constant into a Link register is Ln ← constant. MASS constrains the instruction after the Ln← such that its low 4 bits equal the constant to be loaded and *IA[7]* is set to zero. at's are NOT required.

```
Set[L1.FlagBB, 6];

L1 ← L1.FlagBB,   {loads a 6 into Link1}                    c1;
Noop,                                                        c2{, at[6,10]};
```

2. Loading Link with Conditions: If the microinstruction before the Ln← specifies a branch, dispatch or LnDisp, then the specified bits will be OR'd into the value stored into the Link register. at's are NOT required.

```
[] ← Reg - RegA, NegBr,                                     c1;
L5 ← 2, BRANCH[Pos, Neg],   {Link5 ← IF neg THEN 3 ELSE 2}  c2;
Noop,                                                        c3{, at[2,10]};
```

3. Link Dispatching: LnDisp (equivalently pRetn) is used to dispatch on the value of Link register n. Branches or dispatches can be simultaneously specified. The instruction after an LnDisp must be constrained so that the BRANCH, DISPn or RET has the desired affect. In addition, *IA[7]* of the next instruction must be 1 (MASS does this allocation).

The following example dispatches on (0,,0,,Ureg.8,,Ureg.15) OR Link3 and places the result in Link2:

```
Xbus ← Ureg, L3Disp, XLDisp,                                c1;
L2 ← 0, DISP4[Table],                                       c2;
```

4. Link Subroutines: If Link registers are used for subroutine calling and returning, each subroutine has an associated table of 16 possible return locations. On exit, the subroutine uses a Link register specific to the routine to dispatch into the return table. Thus, a subroutine has a maximum of 16 possible return locations, although it may be called from more than 16 different places if return points are shared. Note that it is possible for a subroutine to have more than 16 return points, and this requires more overhead than the method described here (see below).

The RET macro is used after the pRetn of a subroutine and is equivalent to a DISP4. The CALL macro is used at the subroutine call point and is equivalent to a GOTO.

The Label used in the at macros of the return table does not have to be assigned to any particular return-point instruction, as the following example demonstrates.

L7 ← 0,	c2;
Noop,	c3;
Noop,	c1;
CALL[Sub],	c2;
Noop, {return point 0}	c2, at[0,10,Return];

L7 ← 1,	c1;
CALL[Sub],	c2;
Noop, {return point 1}	c2, at[1,10,Return];

L7 ← 2, CALL[Sub], {only 1 call to Sub can be of this form}	c2;
Noop, {return point 2}	c2, at[2,10,Return];

L7 ← 3,	c2;
CALL[Sub], {call and return point 3}	c2, at[3,10,Return];

Sub:	pRet7,	c3;
	RET[Return],	c1;

{Sub's return dispatch table is
        0: return point 0
        1: return point 1
        2: return point 2
        3: return point 3}

5. Conditional Calls and Returns: Since condition bits can be simultaneously specified with an LnDisp, there can be conditional return points. The same is true of pCalln, so conditional entry points are possible. If the appropriate bits of the return address are not masked, conditional calls always imply conditioned returns (since the condition bits are saved in the Link register).

6. RH Subroutines: RH registers can be used for subroutine calling instead of Link registers. This format requires more microinstruction fields, but is less address-constraining: Calling sequences have no address constraints and return addresses need not have $IA[7]=1$.

rhRet ← 0,	c2;
Noop,	c3;
Noop,	c1;
CALL[Sub],	c2;
Noop, {return point 0}	c2, at[0,10,Return];

rhRet ← 1,	c1;
CALL[Sub],	c2;
Noop, {return point 1}	c2, at[1,10,Return];

rhRet ← 2, CALL[Sub],	c2;
Noop, {return point 2}	c2, at[2,10,Return];

Sub:	Xbus ← rhRet, XDisp,	c3;
	RET[Return],	c1;

7. **Single-Instruction Subroutines:** The following example demonstrates.

```
            MAR ← [rhE, 0 + 0], L6 ← L6.Host0,                   c1;
            L6Disp, CALL[Sub],                                   c2;

            MAR ← [rhE, 1 + 0], L6 ← L6.Host1,                   c1, at[L6.Host0,10,EFetch];
            L6Disp, CALL[Sub],                                   c2;

            MAR ← [rhE, 2 + 0], L6 ← L6.Host2,                   c1, at[L6.Host1,10,EFetch];
            L6Disp, CALL[Sub],                                   c2;

Sub:        EE ← MD, RET[EFetch],                                c3;
```

8. **More than 16 Return Points:** By using RH registers more than 16 return points can be accomodated through multiple return tables. The following example provides 32 possible return locations.

```
Sub:        Xbus ← rhRet, XRefBr,                               c1;
            Xbus ← rhRet, XDisp, BRANCH[Table1, Table2],        c2;
Table1:     RET[ReturnA],                                       c3;
Table2:     RET[ReturnQ],                                       c3;
```

E. SU Registers

The 256 SU (Stack & U) registers are controlled by 3 microinstruction fields: fS, Cin, and $EnSU$. $EnSU$ is true for all SU operations. $Cin=1$ for writes and equals 0 for reads. fS determines how the SU registers are addressed. Cin is also used by the shifts (see sec. C).

1. SU and Arithmetic: When writing SU, Cin must be 1. Therefore, if SU is written and $aF=\{0,1,2\}$, ALU arithmetic must assume a Cin of 1. When reading SU, Cin must be 0. Note that R-R implies $Cin=1$, and R-R-1 implies $Cin=0$.

Because the SU registers are slow sources and slow destinations, they can not be used as an operand in ALU arithmetic nor be written as the result of an ALU arithmetic operation (sec. Q). However, the low byte or nibble, respectively, can be used in arithmetic if less significance is possible.

```
Xbus ← Ureg, Reg ← Reg + RegA,                          c1;
Ureg ← RegA, Reg ← Reg + 1,                             c2;
Ureg ← RegA, Reg ← Reg - Reg,                           c3;
```

2. stackP Addressing: If $fS.2=0$, the SU address comes from the stack pointer $stackP$. The fZ field is free to be interpreted as either $fZNorm$ or a $Nibble$ (but not $IOXIn$, such as ←RH). The macro STK should be used when this addressing mode is desired. See sec. F.

```
STK ← RegA, Reg ← RegA + 0FF + 1,                       c1;
Reg ← STK, rhReg ← RegA LRot0, XLDisp,                  c2;
```

3. U Register Addressing: If $fS.2=1$, the U address is $rA,,fZ$. Since the aS value which combines a U with an R register is D,A and since rA is also used to specify the high four U address bits, a given R register can only be combined (in one microinstruction) with U registers which are in the block of 16 given by the value of the R register. If A-bypass is used in a statement which uses a U register, the same restriction is true.

If $fS[2\text{-}3]=3$, U register addresses must be constrained by the value of the $IOXIn$ source desired. In addition, the rA constraint given above may apply.

```
RegDef[RegA, R, 2];
RegDef[Ureg, U, 2B];    {rA = RegA, fZ = ←RH}
RegDef[Uregx, U, 5B];   {fZ = ←RH}

Ureg ← RegA, Reg ← RHreg, {A-bypass}                    c1;
Ureg ← Reg, Reg ← RHreg, {A-bypass}                     c2;
Uregx ← Q, Xbus ← RHreg, XDisp,                         c3;
```

4. UY Register Addressing: If $fS[2]=1$ and fZ of the previous instruction was AltUaddr, then the U address is $rA,,Y[12\text{-}15]$, where $Y[12\text{-}15]$ results from the previous microinstruction (the same one which contained the AltUaddr). This alternate U register indexing mode can be used to efficiently load a block of 16 U registers from memory (such as from a CSB).

MASS expects a register of type UY, where the 4-bit register number references the block of 16. Since the AltUaddr addressing mode will not properly work across clicks, AltUaddr can not occur in c3, and equivalently, a UY register can not be specified in c1. The following example assumes the 16 words in memory are hex-aligned (rAddr is 0 mod 10).

```
        RegDef[Ublock, UY, 0E];

Cont:   MAR ← [rhAddr, rAddr], rAddr ← rAddr + 1,                       c1;
        [] ← ~0 and rAddr, AltUaddr, NibCarryBr {tests for #0 nibble},  c2;
        Ublock ← rData, rData ← MD, BRANCH[Exit, Cont],                 c3;
```

F. The Mesa Stack & stackP

The Mesa evaluation and argument stack is stored in the bottom SU registers: [0..0F]. When $fS[2]=0$, the 4-bit register *stackP* addresses these locations. The *stackP* can be incremented or decremented independently of the ALU.

1. **Pop and Push:** The pop function decrements the *stackP* by (1 MOD 10) and push increments it by (1 MOD 10). The *stackP* is not updated until after the microinstruction completes.

```
Reg ← STK and Reg, pop,   {uses non-decremented stackP}          c1;
STK ← Reg, push,   {writes STK, then increments stackP}          c2;
```

2. **Reading & Writing stackP:** The *stackP* is loaded from the low 4 bits of the Y bus. One microinstruction must intercede between a **stackP**← and the use of the *stackP* via **STK**.

```
stackP ← Reg + 3,                                                c1;
Noop     {pop or push OK here},                                  c2;
STK ← 0,                                                         c3;
```

The inverse of *stackP* is read via ←ErrnlBnStkp onto X[12-15].

3. **Stack Overflow & Underflow:** To ameliorate checking for stack overflow or underflow, the pop function fields have been asymetrically encoded. The following table shows the allocation of pop's and push's among the function fields. A microinstruction can contain up to 3 pop or push functions: Their effect on the *stackP* and the values of *stackP* which will cause a hardware trap (sec. H) are shown below.

fX	fY	fZ
push	push	push
pop		pop

functions	*stackP*	Trap is	if *stackP* is
pop	-1	underflow	0
push	+1	overflow	15
fXpop, push	0	underflow	0
push, fZpop	0	overflow	15
fXpop, fZpop	-1	underflow	0 or 1
fXpop, fZpop, push	0	underflow	0 or 1

4. **Format of Mesa Stack:** For the Mesa PrincOps evaluation stack, the stack pointer (SP) equals the number of words contained in the stack. Thus, SP = 0 for an empty stack, and SP = StackDepth for a full stack. Also, in the PrincOps stack, the SP indexes the next word above the top of the stack, thus a PrincOps POP must decrement the SP and return the top of stack and a PrincOps PUSH must write first, then increment the SP.

In the Dandelion, the top of stack is kept in an R register called TOS and the word below the top of stack is kept on the top of the "stack" in the SU register file. The SU register which holds stack[SP-1] is called STK and is always pointed at by the contents of the 4-bit *stackP* register. Thus, to pop the Dandelion stack one moves STK to TOS and decrements the *stackP*, and to push one increments the *stackP* and then moves TOS to STK. In order to keep the value of the stack pointers identical for the two stack representations, PrincOps stack locations [1..StackDepth] should be mapped into SU locations [2..StackDepth+1].

Since TOS is a volatile register, it is assumed that TOS can always be saved into SU[*stackP*+1] without any side effects. As an example, if the PrincOps stack has one entry, then TOS is occupied, *stackP*=1, and TOS could be saved in SU[2]. If the stack is empty, then TOS is empty, *stackP*=0, and TOS could be saved in SU[1].

Stack overflow is defined to be $stackP=$ StackDepth + 1 and push. This implies that whenever a true PrincOps PUSH is desired, the $stackP$ must be push'd twice and pop'd once. In general, $stackP$ can always be incremented once (to save TOS into STK) without fear of overflow, but if we are truly putting one more word on the Mesa stack, it must be incremented once more.

Stack underflow occurs when $stackP=0$ and a pop is attempted.

The maximum value of StackDepth is 14: overflow at $stackP=15$ and push, underflow at stackP = 0 and pop. Currently, the hardware PROM which checks the stack size assumes a maximum stack of 14 words.

5. Preserving TOS: In general, the previously-executed Mesa instruction may complete executing without duplicating TOS into STK. Therefore, each Mesa instruction implementation must, if neccessary, save TOS into SU[$stackP+1$] before it modifies TOS. According to PrincOps, if TOS is an argument to the bytecode, TOS should be saved (so it can be recovered by a Mesa PUSH) if either the Mesa bytecode does not change the contents of the stack or does not change the value of SP. SLn and JEQn are two examples.

Note that as a part of normal stack maintenance, TOS must be saved into STK if the Mesa opcode implementation is pushing data onto the stack.

G. Instruction Buffer & PC16

The instruction buffer (IB) holds bytes from the code segment while a Mesa opcode is executing. The 3-byte buffer guarantees that the opcode will always find its arguments in the buffer, thereby eliminating the need for another type of trap. Buffer refill traps occur only between the execution of Mesa opcodes when the buffer is not full.

The state of the buffer is given by the instruction buffer pointer *ibPtr*, which is automatically updated by the functions which read and load the buffer. The Mesa byte program counter, which is contained in the R register **PC** and the 1-bit extension *PC16* register, must be maintained independently and in synchrony with the *ibPtr*.

The IB holds a maximum of 3 bytes, the minimum number necessary to complete a Mesa instruction. Whenever a Mesa opcode completes (executes an **IBDisp**) and there are not 3 bytes in the buffer, a microcode trap is caused which results in refilling of the IB. The so-called "refill" microcode executes in one click if 2 more bytes are needed and in two clicks if 4 are needed. The refill code also dispatches on the first arriving memory byte or the front of the buffer, so if 4 bytes were fetched, 3 are retained.

The bytes in the IB are ordered by their arrival from the X bus: the most significant byte is placed "before" the least significant byte. The first byte in the IB is called *ibFront*, the second is *IB[0]* and the third is *IB[1]*.

1. **Instruction Buffer States:** The IB has four possible states as given by the 2 bit *ibPtr* register:

state name	bytes in IB	ibPtr
full	3	2
word	2	3
byte	1	1
empty	0	0

Note that the complement of *ibPtr* is read by ←**ErrnIBnStkp**, and appears on X bus bits [10..11].
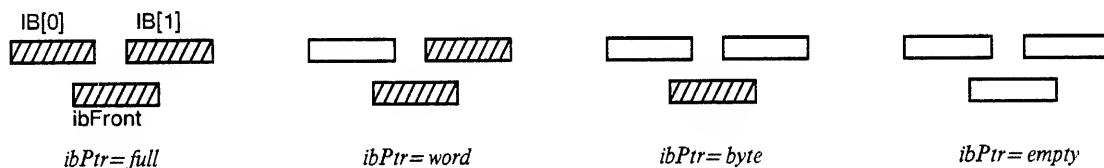
Figure 4. Instruction Buffer States

2. **Instruction Buffer Functions:** The following table summarizes the effect of the various IB related function fields on the *ibPtr*, *ibFront*, and the X bus. Figure 5 graphically shows sequences of typical IB functions.

function	new *ibPtr*	new *ibFront*	Xbus ←
←ib	ibPtr-1	IB[ibPtr[1]]	0,,ibFront
←ibNA	unchanged	unchanged	0,,ibFront
←ibHigh	unchanged	unchanged	0,,ibFront[0-3]
←ibLow	unchanged	unchanged	0,,ibFront[4-7]
IBDisp	ibPtr-1	IB[ibPtr[1]]	not affected
AlwaysIBDisp	ibPtr-1	IB[ibPtr[1]]	not affected
IB←	IF *empty* THEN *word* ELSE *full*	IF ibPtr=*empty* THEN X[0-7] ELSE unchanged	not affected
IB←, ibPtr←1	IF *empty* THEN *byte* ELSE *full*	IF ibPtr=*byte* THEN X[8-15] ELSE unchanged	not affected
IBPtr←0	*word*	IB[0]	not affected
IBPtr←1	*byte*	IB[1]	not affected

←ib, IBDisp, and AlwaysIBDisp cause the *ibPtr* to "decrement" by 1. The *ibPtr* "counts" *full, word, byte, empty,* or 2, 3, 1, 0.

IB← causes *ibPtr* to be set to *word* if it was *empty* and otherwise sets it to *full*. When the buffer is empty, this convention sets *ibFront* to the left half of an incoming word. If IBPtr←1 is executed along with the IB←, *ibFront* will get the right half of an incoming X-bus word.

IBPtr←0 sets *ibPtr* to *word* and IBPtr←1 sets it to *byte*.

←ibNA, ←ibHigh, and ←ibLow do not change *ibPtr*.

IB[0],,IB[1] is parallel-loaded by a word from the X bus via IB←. *IB[0]* gets X[0-7] and *IB[1]* gets X[8-15].

ibFront is loaded via IBDisp, AlwaysIBDisp, ←ib and IB←. IB← only loads *ibFront* if the old *ibPtr=empty*. When it is loaded, its value comes from *IB[0]* if the old *ibPtr* was *full* and from *IB[1]* if the old *ibPtr* was *word*. When *ibFront* is loaded by IBPtr←n, its value comes from *IB[n]*.

3. **Reading Instruction Buffer:** ←ib and ←ibNA cause *ibFront* to be placed onto the low half of the X bus, while ←ibHigh puts the high 4 bits, and ←ibLow the low 4 bits of *ibFront* onto X[12-15]. High order X-bus bits are zero'd. ←ibNA does not advance the *ibPtr*.

```
        MAR ← [rhL, L + ib],                        c1;
        rhT ← ibLow,                                c2;
```

4. **Dispatching on IB:** IBDisp and AlwaysIBDisp cause a 256-way dispatch based on the value of *ibFront*. They can only be specified in c1 or c2, but, by convention, they are restricted to c2 so that the first instruction executed after the dispatch is in c1. (The IBDisp can not be in c3 because only 4 dispatch bits are saved across clicks.)

The high 4 bits of *ibFront* replace *INIA[4-7]*, while the low 4 bits of *ibFront* are OR'd with *INIA[8-11]* (thereby allowing simultaneous branch/dispatches). *INIA[0-3]* is unaffected, so there are 16 possible 256-way dispatch tables which can be simultaneously specified.

5. **Instruction Buffer Refill Traps:** If an IBDisp is executed and *ibPtr≠full*, the dispatch does not occur and instead a microcode trap is caused. *INIA[0-3]* is replaced with 4 when *ibPtr=empty* or 5 when *ibPtr≠empty*. If there is a pending Mesa interrupt request, *MInt=1*, *INIA[0-3]* ← 6 (if the IB is also empty) or *INIA[0-3]* ← 7 (if the IB is also not empty). If either trap occurs *ibPtr* does not change. The Error microcode trap to location 0 has priority over the IB traps (see sec. H).

Since *INIA[4-11]* (the low 8 bits of OpTable in the example below) will normally be zero, IB traps will occur according to this table:

Trap	location (hex)
IB empty	400
IB not empty	500
Mesa interrupt	600 or 700

6. Non-trapping IB Dispatch: AlwaysIBDisp acts like an IBDisp but does not trap: AlwaysIBDisp will dispatch on *ibFront* even if there is a pending Mesa interrupt request or the buffer is not full. (It is used in the refill code). AlwaysIBDisp is encoded by *fY*=IBDisp and *fZ*=IBPtr←1 and is defined as a macro in Dandelion.df.

7. pageCross Cancel of IB Dispatch: If the microinstruction executed before either an IBDisp or AlwaysIBDisp is a MAR← which causes a *pageCross* branch, the instruction buffer dispatch will not occur and *ibPtr* will remain unchanged. Control will go to location *INIA* instead (label OpTable in the example below).

8. DISPNI macro: The DISPNI macro is used after IBDisp's and AlwaysIBDisp's. It is equivalent to the GOTO macro except it zero's the low 8 bits of its argument address.

```
MAR ← [rhPC, PC],                          c1;
AlwaysIBDisp, BRANCH[$, Cross, 1]          c2;
IB ← MD, DISPNI[OpTable],                   c3;
```

9. IB Empty Error Trap: An Error Trap to location 0 occurs if a ←ib, ←ibNA, ←ibLow, or ←ibHigh is executed with *ibPtr=empty*. This trap has priority over the IB refill-interrupt trap described above (although *ibPtr* will still change if they occur simultaneously).

This trap is used at code segment page crossings to verify whether control will actually proceed to the next sequential page so that a possible unwarranted page fault can be avoided. It is not necessary to use this trap if the software can tolerate occasional page faults to code segments which actually aren't needed. In particular, if this trap is not utilized, a fault may be caused to the (non-existent) page following a code segment which ends exactly on a page boundary.

This trap is utilized as follows: If during the IB not-empty refill, the Mesa program counter points to the last word of a page, instead of being refilled, the IB is left untouched (*ibPtr≠full*) and control is returned to execute the next Mesa bytecode. If this byte code uses code segment bytes which straddle the page boundary, an IB Empty Error trap will occur, and only then will the next code segment page be mapped (possibly resulting in a page fault).

Note that ←ib, ←ibNA, ←ibLow, or ←ibHigh references never occur in c2 or c3 of the last click of a Mesa opcode. Therefore, since the IB Empty Error trap can only occur in c1 of the last click or before the last click, the trap is guaranteed to occur before control dispatches to the next opcode.

In order for this trap to properly work, the refill code which executes when the PC points to the last word of the page must save the appropriate Mesa PrincOps state so that it can be restored after the opcode traps (if indeed it does). Thus, TOS, PC, *pc16*, and the *stackP* must be saved. L, G, rhMDS, UvL, UvG, etc. need not be saved since, if they are changed, control is going elsewhere anyway.

Since main memory is obviously part of the Mesa PrincOps state, IB references (←ib, ←ibNA, ←ibLow, or ←ibHigh) can not occur during or after the first MDR← in the opcode implementation. (In particular, an incorrect memory location may be written if the memory address is indexed by an IB byte). Since the IB reference can occur in c1 (such as in the SLB opcode), the hardware cancels a memory write if the IB Empty Error trap occurs in c1.

10. PC16: *pc16* is a 1-bit register which is utilized as an extension to the Mesa program counter. The word part of the counter is stored in the R register **PC** and the byte index in *pc16*.

If *fX* or *fZ* is *Cin←pc16*, *pc16* becomes the *Cin* for the ALU. Also, *pc16* is inverted at the end of the instruction. Thus, the following statements are used to add or subtract from the byte program counter **PC**,*pc16*.

        PC ← PC + PC16, {adds 1}   c1;
        PC ← PC - PC16, {subtracts 1}   c2;
        PC ← PC + 2 + PC16, {adds 5}   c3;

Since *Cin* is also the shift ends, **SE←pc16** can be used to load *pc16* into an R register, thereby reconstructing the Mesa program counter in one microinstruction.

        PC ← LShift1 PC, SE←pc16,   c1;

The **XC2npcDisp** OR's the inverse of *pc16* into *INIA.11*.

Due to the way *Cin* is implemented in the hardware, when the *Cin* field of the microinstruction is 0, the *fX* version of *Cin←pc16* must be used. (If the *fZ* version is used, *Cin* will be 0 instead of *pc16*.) If *Cin=1*, then either version of *Cin←pc16* can be used. (Note that *Cin=0* on read SU, and *Cin=1* for write SU.) MASS guarantees these assignments.

	c2	c3	c1	c2	c3

Empty Refill   IB←MD   AlwaysIBDisp   IB←MD

1 byte remaining Refill   AlwaysIBDisp   IB←MD

2 bytes remaining Refill   AlwaysIBDisp   IB←MD

Jump to Even Byte   ibPtr = ??   IB←MD   IBPtr←0   AlwaysIBDisp   IB←MD

Jump to Odd Byte   ibPtr = ??   IB←MD   IBPtr←1   AlwaysIBDisp   IB←MD

Full,IBDisp   IBDisp

Full,←ib, IBDisp   ←ib   IBDisp   GOTO[2 bytes Refill]

Full,←ib, ←ib,IBDisp   ←ib   ←ib   IBDisp   GOTO[1 byte Refill]

Full,←ib, ←ib,←ib, IBDisp   ←ib   ←ib   ←ib   IBDisp   GOTO[Empty Refill]

Figure 5. Typical Instruction Buffer Sequences

H. Control Store Traps

There are two general classes of control store traps: The Instruction Buffer Refill traps (described in sec. G.5) and the so-called Error traps.

The Error traps cause control to be transfered to control store location 0 which will execute in c1. The *EKErr* (*E*mulator-*K*ernel *Err*or) register indicates the type of error:

EKErr	Error
0	Control Store Parity
1	Emulator Virtual Address Out of Range (>22 bits) OR Emulator Double Bit Memory Error
2	*stackP* Overflow or Underflow
3	IB Empty Error (see sec. G.9)

Smaller values of *EKErr* have priority over larger values. Thus, if a control store parity error occurs along with a double bit memory error, then the CS parity error will be indicated instead of the memory error. The errors also accumulate, so that if a higher priority error occurs after a lower one has been detected (but before *EKErr* has been read), the higher priority error will be indicated. Also note that these Error traps have priority over the IB Refill trap.

EKErr is read by the ←ErrnIBnStkp function onto X[8-9].

1. CS Parity Error: If a parity error occurs (i.e., odd parity) while any task is executing, then control will go to location 0 at the Kernel task level. Since the Kernel task is of highest priority, no other tasks will execute. The trap code should turn off IO device write operations.

This error is non-recoverable (and unfortunately depends on the reliability of the parity-&-error-reporting circuits). The CS Parity Error signal is available to the IO Processor (IOP), so that a system boot (or whatever) can take place.

Since the default value of *EKErr* is 0, if control inadvertently reaches CS location 0, this will appear to be a CS parity error (except to the IOP).

Timing: If the CS parity error occurs in c1 (meaning the instruction which would have executed in c2 has odd parity), then the next click will be the Kernel's at location 0. If the error occurs in c2 or c3, one more click will execute before control goes to location 0.

2. Memory Error: If *EKErr=1*, then a memory error of some type occured. If bit 8 of the Memory Status register, *MStatus[8]*, is set, then an Emulator double bit memory error was the cause of the trap. (This does not detect IO task double-bit errors. See sec I.3)

If *MStatus[8]=0*, then a virtual address used during a Map← was out of range. This Emulator trap occurs for ANY task which executes Map←'s! Currently, the hardware PROM assumes virtual addresses are 22 bits wide, so if either of the 2 high-order bits of an RH register used in a Map← are non-zero, then a trap will occur.

This mechanism places Emulator double bit memory errors at higher priority than virtual address out-of-range errors. It also assumes that the Memory system error-logging register was reset (see sec. I.3).

Timing: If a memory error occurs (virtual address error in c1 or double-bit error in c3), at most one additional Emulator click can execute before control is at location 0 in c1 at the Emulator task level.

3. Stack Error: If a pop or push is executed with the values of *stackP* given in sec. F.3, then a Stack Error trap occurs. The *stackP* will still be incremented or decremented if the error occurs. The hardware PROM which checks the *stackP* can be reprogrammed to assume various maximum stack sizes up to 14 words.

Timing: If a stack trap occurs (any cycle) at most one additional Emulator click can execute before control is at location 0 in c1 at the Emulator task level.

4. IB Empty Error: See section G.9.

5. ClrIntErr: ClrIntErr resets both *MInt* and *EKErr*. Therefore, *MInt* should be set if ClrIntErr is being used to reset *EKErr*.

Executing a ClrIntErr means that the trap to location 0 will not repeat unless the condition which caused the trap is still present (i.e., CS parity error) or a new hardware trap occurred.

6. Example Error-Trap-Catching Code: The following is typical location-0 code. Every stand-alone program MUST have at least the instruction shown at location 0.

```
ErrTrap:    T ← RRot1 ErrnIBnStkp, ClrIntErr, CANCELBR[$, 0F],      c1, at[0];
            Xbus ← T LRot0, XwdDisp,                                 c2;
            DISP2[ErrType],                                          c3;

ErrType:    KCtl ← 0, GOTO[CSParErr],                                c1, at[0,4,ErrType];
            Xbus ← MStatus, XLDisp, GOTO[MemErr],                    c1, at[1,4,ErrType];
            GOTO[StackErr],                                          c1, at[2,4,ErrType];
            GOTO[IBEmptyErr],                                        c1, at[3,4,ErrType];

MemErr:     BRANCH[VirtAddrErr, EmuMemErr, 1],                       c2;
```

I. Memory

The memory system accepts either real addresses (via MAR←) or virtual addresses (via Map←). When virtual addresses are specified, the memory returns as data the real page number which corresponds to the virtual page part of the virtual address and some flags (See sec. J on Mapping). This section will deal with real address accesses only.

Currently, the memory address register (MAR) is 18 bits wide and the memory size is 192K words. Assuming the current format of Map entries, the maximum possible address width is 20 bits, which implies a maximum real memory size of 1M words.

Addresses are sent to the memory system via the YH and Y buses, where YH is the output of the RH register used to hold the 2 high-order address bits. Thus, YH[6],,YH[7],,Y[0-7] holds the page number and Y[8-15] holds the page displacement.

The *mem* bit of the microinstruction format is used to designate memory operations. If *mem* is set and the instruction is executing in c1, MAR will be loaded from YH,,Y. If set in c2, the memory write data register (MDR) will be loaded from the Y bus and the memory location will be written. If *mem* is set in c3, returning memory data (MD) will be placed onto the X bus.

1. Loading MAR: MAR ← [rhReg, ⟨arithPhrase⟩] designates a real-address reference to memory and can only occur in c1.

rhReg specifies the RH register which holds the 2 high-order address bits, and, with some formats, which R/RH register pair holds the 10 high-order address bits. The *rB* field is set to the value of rhReg. Notationally, ⟨arithPhrase⟩ is anything that can occur on the right side of an arithmetic clause.

An RH register should not be loaded in the same instruction which specifies a MAR←. Due to timing, MAR can only be loaded with a constant, the IB, or Errn|Bn|Stkp from the X bus (sec. Q).

2.   MAR← Side Effects: There are three important side effects of a MAR←:

(a)   MAR← forces $aS{\leftarrow}0,B$ and $aF{\leftarrow}aF\ OR\ 3$ for the high half of the ALU;
(b)   MAR← enables a *pageCross* branch in *INIA[10]*; and
(c) If a MAR← causes a *pageCross* branch to occur, a following MDR←, IBDisp, or Always|BDisp will be cancelled.

(a) *aF←aF OR 3* for ALU[0-7]: If the *aF* field of the MAR← instruction is $R+S$, $S{-}R$, $R{-}S$ or *RorS* (i.e., 0..3), *aF* for the high half of the ALU (bits 0-7) will be set to *RorS*, causing the ALU output F[0-7] to equal the high half of the register given by *rB*. MASS currently does not allow *aF* values of [4..7] during a MAR←.

In general, if A-bypass is not used, the upper 10 bits of the memory address (i.e., the page address) come from the RH/R pair given by the *rB* field, while the lower 8 bits (i.e., the displacement within a page) come from the source defined by ⟨arithPhrase⟩. This feature is used to combine the real page number (as read from the Map) with a displacement into the page in one cycle. (This method also meets memory system timing requirements.)

For example,

```
MAR ← R ← [rhR, RA + 2],        c1;
```

causes

```
YH   ←   rhR,
Y[0-7]   ←   R[0-7]   ←   R[0-7],
Y[8-15]   ←   R[8-15]   ←   RA[8-15]   +   2.
```

If A-bypass is used, MAR receives the complete R register given by the rA field, but the high half of the ALU is still affected and the high 2 bits of address still come from RH[rB]. A-bypass also implies that the R register given by rB must be written. Thus,

```
MAR ← [rhR, RA], R ← RA +2,      c1;
```

causes

```
YH   ←   rhR,
Y   ←   RA,
R[0-7]   ←   R[0-7],
R[8-15]   ←   RA[8-15]   +   2.
```

A consequence of forcing $F[0\text{-}7] \leftarrow R[rB]$ is that the carry out from the low ALU half does NOT propagate into the high half. Thus, after the following statement is executed, R[0-7] is unaltered:

```
MAR ← R ← [rhR, R + OFF + 1],                              c1;
```

(b) pageCross: The second major effect of a MAR← is that it automatically specifies a *pageCross* branch. A branch will occur in *INIA[10]* if the evaluation of the ALU operation (e.g., <arithPhrase>) results in a carry from the low byte. This usually implies that a remapping of the real address is necessary.

pageCross is defined to be *pageCarry* XOR $aF.2$, where *pageCarry* is the carry out of the low 8 ALU bits. This has the effect of toggling *pageCarry* when doing subtraction ($aF = S\text{-}R$). *pageCross* equals *pageCarry* when doing addition ($aF = R + S$). Thus, if you use positive displacements, such as Reg + 1 or Reg-1, *pageCross* will consistently indicate when a page boundary has been crossed.

However, the $aF = R\text{-}S$ form of subtraction, unlike $aF = S\text{-}R$, does NOT cause *pageCarry* to be toggled on subtraction (since $aF.2 = 0$). However, the $aF = S\text{-}R$ form covers most of the common subtraction cases: B-1, A-1, B-A, A-constant, and Q-constant. It does NOT include D-1. MASS will check whether $aF = R\text{-}S$ has been used, and if so desired, LOOPHOLE[pci] ("PageCross Inverted") will allow its application.

The *pageCross* branch occurs in *INIA[10]*; therefore the BRANCH[Label0, Label1, 1], CANCELBR[Label, 2], or DISPn[Label] forms must be used after a MAR←. Since the branch occurs in *INIA[10]*, other branch conditions can be simultaneously specified, as the following example shows:

```
        MAR ← Reg ← [rhReg, Reg + 1], ZeroBr,                     c1;
        MDR ← Ureg, DISP2[Table],                                 c2;

Table:  GOTO[NotZero],                                            c3, at[0,4,Table];
        GOTO[Zero],                                               c3, at[1,4,Table];
        GOTO[PgCross&NotZero],                                    c3, at[2,4,Table];
        GOTO[PgCross&Zero],                                       c3, at[3,4,Table];
```

The implied *pageCross* branch can be canceled by CANCELBR[Label, 2] or DISPn[Label, 2]. If it is known that the ALU arithmetic operation will never produce a *pageCross*, CANCELBR[Label, 0] can be used. This will reduce the number of control store restraints.

Since the ALU can generate *pageCarry's* on logical operations, one should avoid the logical functions (*aF = RorS* in particular) during a MAR←. This serves to reduce the number of allocation constaints and may be necessary for memory writes (see below).

For example, you must explicitly write MAR ← [rhR, R + 0], where MAR ← [rhR, R] was considered. In general, if MASS doesn't see an expression of the form register + 0, then the MAR← must be followed by either BRANCH, DISPn, or CANCELBR. If *Cin←pc16* is present with MAR←, then a *pageCross*-caused branch is always possible.

The following forms do NOT require a CANCELBR (and will not cancel an MDR←). The final item is allowed because " + 0" timing is a special case.

```
MAR ← [rhR, RA + 0],                                        c1;
MAR ← [rhR, RA], R ← R + 0,                                 c1;
MAR ← [rhR, 0 + 0], {first location of page}               c1;
MAR ← [rhR, 0 + 0FF], {last location of page}              c1;
MAR ← [rhR, RA], R ← Ureg + 0, LOOPHOLE[byteTiming]        c1;
```

(c) Cancelation of MDR← & IBDisp: If a *pageCross* branch actually occurs during a MAR←, then a following MDR←, IBDisp, or AlwaysIBDisp is canceled. This prevents writing into the wrong memory page (if A-bypass was not used) or dispatching on the next Mesa instruction if a page crossing has been indicated.

This feature makes it manditory that logical functions not be used during a MAR← (see above).

This characteristic of *pageCross* is easily overlooked, so to catch a common misunderstanding, MASS disallows CANCELBR[Label, 2] with an MDR←. However, if you know that when the *pageCross* actually occurs the MDR← SHOULD be canceled, then the macro LOOPHOLE[wok] (also called WriteOK in Dandelion.df) will make MASS happy. In the following example, the write will be canceled when Line = 0FF.

```
MAR ← [rhLine, Line + 1],                           c1;
MDR ← dX, LOOPHOLE[wok], CANCELBR[$, 2],            c2;
Noop,                                               c3;
```

3. Display Bank Contention: If a MAR← is executed in clicks 0 through 3 (i.e., all clicks except for the Display click) and the Display controller hardware is reading from the display bank (low 64K of memory), the click will be aborted. In other words, none of the microinstructions of the click will be executed and the machine state will remain unchanged.

4. Error Correction: The memory system corrects single-bit errors and detects double-bit errors read from any bank. A memory read to non-existent memory causes a double-bit memory error.

(a) Erorr-Logging Register: Associated with each task is a 1 bit register which is set whenever a double-bit memory error occurs on an ←MD executed by the task. The 8-bit error register is loaded onto X[8-15] by ←MStatus, where bit 8 is task 0's log bit and bit 15 is task 7's log bit.

Individual bits in the logging register can be reset by executing an MCtl← with Y[4] = 1 and Y[5-7] = task # for the logging register to be reset.

IO tasks, before entering a critical data transfer phase can clear and then check the task's error bit after moving the data. Note that there is special microcode trap if the Emulator task causes a double-bit memory error (see Sec. H).

(b) MCtl & MStatus: If *MCtl* is loaded with Y[15]=1, future single bit errors will not be corrected. If any of Y[8-13] is set, the corresponding check bit written into memory will be inverted. This is used for verifying the correction system.

On a ←MStatus, X[6] indicates a single-bit error and X[7] indicates whether a double bit error occured on the previous memory read. X[0-5] are the syndrome bits of the last read.

←MStatus:

Syndrome								Error Log							
A	B	C	D	E	F	se	de	t0	t1	t2	t3	t4	t5	t6	t7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

MCtl←:

			clr	task		A	B	C	Inv Check D	E	F			en	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

5. Memory Notes:

(a) Due to the A-bypass mechanism, a memory address can be incremented either before or after being sent to *MAR*. However, the latter case, as part of a writing loop, prevents the last word of a page from being stored.

```
MAR ← Reg ← [rhReg, Reg + 1],                              c1;
MAR ← [rhReg, Reg], Reg ← Reg + 1,                         c1;
```

(b) The implied *pageCross* branch can be used to indicate the end of a count sequence. Cnt[8-15] is initialized with 256-actualCount, rhCnt[6-7],,Cnt[0-7] holds the page address, and Reg[8-15] holds the location within the page. If the MAR← is part of a writing loop, the associated MDR← will be canceled on exit.

```
MAR ← [rhCnt, Reg], Cnt ← Cnt + 1,                         c1;
```

(c) If a memory location is both read and written in the same click, the old contents are returned.

```
MAR ← [rhR, R + 0],                                        c1;
MDR ← New,                                                 c2;
Old ← MD,                                                  c3;
```

(d) Due to a design bug in the current memory controller, IO tasks should not execute either an MDR← or ←MD without a corresponding MAR← (or Map←). However, these are legal for the Emulator so long as IO microcode does not read or write the display bank when the display is on.

MASS checks for this and LOOPHOLE[mdrok] or LOOPHOLE[mdok] can be used appropriately.

The memory bug causes all display-contention-aborted clicks to be memory-cycle clicks. However, in general, if there is not a MAR←, a double-bit error will not occur on a ←MD and memory will not be written by an MDR←.

J. The Mesa Map

The Map is a 16K-word table located after the display bank in real addresses [10000..13FFF]. The memory system indexes into this table with the 14-bit page part of a 22-bit virtual address. Entries contain a 12-bit real page number (only 10 bits currently used) and some flags pertaining to the virtual page. Figure 6 illustrates the mapping process.

The Map assumes a 22-bit virtual address space (4M words), but a smaller Map is possible since all 16K does not have to be used. Currently, 23- or 24-bit virtual addresses will result in a hardware trap (sec. H), but a hardware PROM (and the memory system) can be changed to allow larger virtual addresses.

Virtual addresses are sent to the memory system via the YH and Y buses, where YH is the output of the RH register used to hold the 6 high-order address bits. Thus, YH[2-7],,Y[0-7] holds the page number and Y[8-15] holds the page displacement. Note that Y[8-15] is ignored by the memory. If YH[0] or YH[1] is set, the virtual-address-out-of-range microcode trap is caused (sec. H).

If either fX or fY is set to $Map\leftarrow$ (in c1), the memory system will assume a Map reference for the entire click. If the *mem* bit of a microinstruction is set in c2 of the click, the memory write data register (*MDR*) will be loaded from the Y bus and the Map entry will be written. If *mem* is set in c3, returning Map data (MD) will be placed onto the X bus.

The *mem* bit should not be set in c1 along with the Map← unless MAR←'s side effects are explicitly desired. This is accomplished by writing Map ← MAR ←. Note that such clicks could be aborted due to display bank contention (sec. I.3).

1. Map←: Map ← [rhReg, <arithPhrase>] designates a virtual address reference to the Map and can only occur in c1.

rhReg specifies the RH register which holds the 6 high order address bits. The *rB* field is set to the value of rhReg. Notationally, <arithPhrase> is anything that can occur on the right side of an arithmetic statement.

An RH register should not be loaded in the same instruction which specifies a Map←. Due to timing, Map← can not be loaded from an arithmetic operation which uses an X-bus operand.

```
Map ← [rhR, R],                        c1;
Map ← R ← [rhR, RA + 1],                c1;
Map ← [rhR, RA], R ← RA + OFF + 1,      c1;
```

2. Map Entry Format: All entries of the Map are of the form:

page[4-11]	dp	w	d	rp	page[0-3]
0	8	9	10	11	12 15
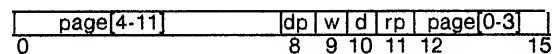
Field	Bits	Function
page	12-15,,0-7	real page number (12 bits)
dp	8	dirty & present
w	9	write protected
d	10	dirty
rp	11	referenced & present

The Map flags have the following interpretation, where *present* is a derived column and is the inverse of the PrincOp's *vacant*. The PrincOp's *referenced* bit is equivalent to the *rp* bit here.

w	d	rp	dp	present	Interpretation
0	0	0	0	1	untouched, unprotected
0	0	1	0	1	unwritten, read
0	1	0	1	1	reserved for software
0	1	1	1	1	written
1	0	0	0	1	untouched, protected
1	0	1	0	1	protected, read
1	1	0	0	0	vacant
1	1	1	0	1	reserved for software

3. Maintaining the Map Flags: The Map flags must be checked with every virtual address access. They only need to be updated the first time a present virtual page is either read or written. Thus, a normal virtual address access requires two clicks: one for the Map read and one for the real address access. As a consequence, the microcode typically saves the real page number acquired in the first Map reference and then reuses it until invalid, such as with a page-crossing.

Because a Map entry can not be updated atomically, IO task microcode can not maintain the Map flags, i.e., only the Emulator can write into the Map.

(a) Virtual Read: When a Map← is executed with the intent of reading, XRefBr is used to branch on the *rp* bit returned in c3. If *rp=0*, then either the page has not been previously referenced or it is not present. If *rp=1*, the microcode can proceed assuming that the page is present in real memory and no Map maintenance is required.

When *rp* is 0, the microcode can XwdDisp on *w,,d* and act according to the following table:

w	d	action
0	0	rewrite Map entry with *rp←1* & continue
0	1	rewrite Map entry with *rp←1* & continue
1	0	rewrite Map entry with *rp←1* & continue
1	1	restore Mesa state & Xfer to SD[sPageFault]

(b) Virtual Write: Similarly, when a Map← is executed with the intent of writing, XDirtyDisp (XLDisp) is used to branch on the *dp* bit returned in c3. (Note that the branch occurs in *INIA[10]*.) If *dp=0*, then either the page has not been previously written, or it is write-protected, or it is not present. If *dp=1*, the microcode can proceed assuming that the page is present and writeable and no Map maintenance is required.

When *dp* is 0, the microcode can XwdDisp on *w,,d* and act according to the following table:

w	d	action
0	0	rewrite Map entry with *d←dp←1* & continue
0	1	rewrite Map entry with *d←dp←1* & continue
1	0	restore Mesa state & Xfer to SD[sWriteProtect]
1	1	restore Mesa state & Xfer to SD[sPageFault]

Note that for Page and WriteProtect Faults PC,,*pc16*, *stackP*, TOS, STK, and any other altered Mesa state must be restored to its value at entry to the Mesa instruction. (Link registers are typically used to encode the required state fixup.)

The following example of write-Map update code is the body of the W2 Mesa opcode. The common subroutine **WMapFix** returns through Link register 0 if the page is present and through Link 1 if the page is swapped out. This code also shows the saving of TOS into STK (sec. F.5), poping the address & data from the stack, encoding state fixup in Link registers and dispatching on the next opcode.

```
@W2:        Map ← Q ← [rhMDS, TOS + 2], L1←L1wDecOnly,           c1, opcode[111'b];
            PC ← PC + PC16, push, L0←L0.RedoW,                   c2;
            Rx ← rhRx ← MD, XDirtyDisp, STK ← TOS, pop,          c3;

RedoW:      MAR ← [rhRx, Q + 0], BRANCH[WxMapUD, $, 1],          c1, at[L0.RedoW,10,WMapFixCaller];
            MDR ← STK, pop, IBDisp,                              c2;
            TOS ← STK, pop, DISPNI[OpTable],                     c3;

WxMapUD:    CALL[WMapFix] {will return to RedoW},                c2;


{Write Map Update Subroutine}
WMapFix:    Xbus ← Rx LRot0, XwdDisp, L3←L3.rhMDS.Q,             c3;
            Map ← [rhMDS, Q], DISP2[FixWFlags],                  c1;

FixWFlags:  MDR ← Rx or 0B0, L0Disp, GOTO[ReWrite],             c2, at[0,4];
            MDR ← Rx or 0B0, L0Disp, GOTO[ReWrite],             c2, at[1,4,FixWFlags];
            T ← sWriteProtect, L1Disp, GOTO[WTrap],             c2, at[2,4,FixWFlags];
            T ← sPageFault, L1Disp, GOTO[WTrap],                c2, at[3,4,FixWFlags];

ReWrite:    Xbus ← 2, XDisp, RET[WMapFixCaller],                c3;
```
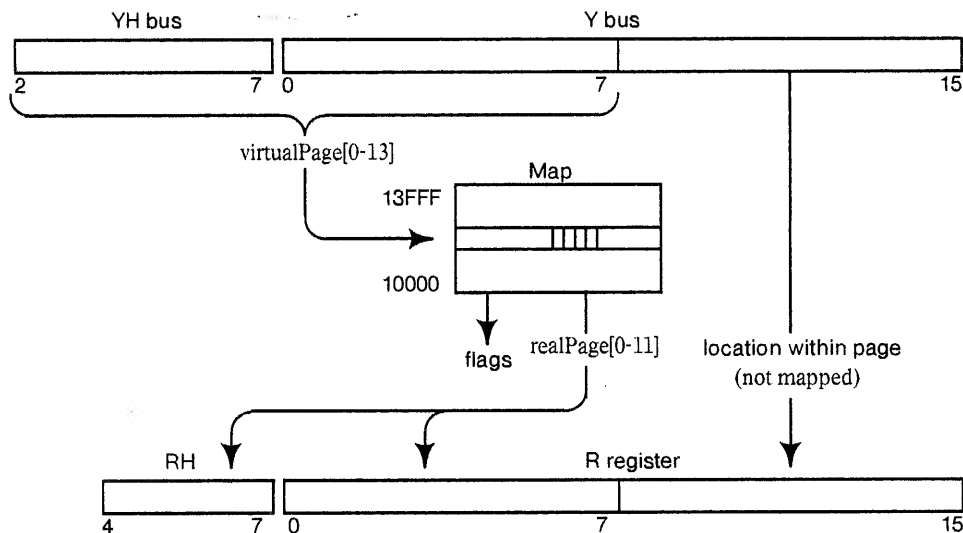

Figure 6. Map Reference

K. Bus Destinations

All possible X- and Y-bus destinations are included in this list. The non-IO registers have been discussed elsewhere.

Destination	Source Bus	
MAR←	YH,,Y	Memory Address Register (for real address)
Map←	YH,,Y	Memory Address Register (for Map reference)
MDR←	Y	Memory Data Register
STK←	Y	SU register addressed by stackP
stackP←	Y	stack Pointer
Ureg←	Y	U register
DCtlFifo←	Y	Display Output Data Control Fifo
DBorder←	Y	Display Output Data Border Pattern register
MCtl←	Y	Memory Control register
IB←	X	Mesa Instruction Buffer
RHregister←	X	RH registeristers
KOData←	X	Rigid Disk Output Data register
EOData←	X	Ethernet Output Data register
POData←	X	Printer Output Data register
IOPOData←	X	IOP Output Data register
KCtl←	X	Rigid Disk Control register
KCmd←	X	Trident Disk Control register
EICtl←	X	Ethernet Input Control register
EOCtl←	X	Ethernet Output Control register
IOPCtl←	X	IOP Control register
DCtl←	X	Display Control register
PCtl←	X	Printer Control register

L. X Bus Sources

All possible X-bus sources are listed here; the non-IO registers have been discussed elsewhere. Xbus[0-7] is set to zero with *Nibble, Byte,* or *IOXIn* = {8..0F} (marked * in list). Also, Xbus[8-11] is set to 0 on ←ibLow, ←ibHigh and *Nibble* (marked ** in list}. Even though KStrobe and EStrobe are listed under *IOXIn*, they are not X-bus sources.

X bus Source

← Y LRotn	Left Rotate Y bus by 0, 4, 8, or 12 positions
fZ	** 4 bit constants (Nibble)
fY,,fZ	* 8 bit constants (Byte)
←MD	Memory Data
←STK	SU register addressed by stackP
←Ureg	U register
←RHreg	* RH registers
←ib	* Instruction Buffer
←ibNA	* Instruction Buffer (doesn't advance IBPtr)
←ibLow	** ib[4-7]
←ibHigh	** ib[0-3]
←KIData	Rigid Disk Input Data
←EIData	Ethernet Input Data
←IOPIData	* IOP Input Data
←ErrnIBnStkP	* X[8-9] = EKErr, X[10-11] = ~ibPtr, X[12-15] = ~stackP
←KStatus	Rigid Disk Status
←KTest	Rigid Disk Status (diagnostics)
←EStatus	Ethernet Status
←MStatus	Memory Status
←IOPStatus	* IOP Status

K. Miscellaneous Functions

1. **MesaIntRq:** This function sets the 1-bit Mesa Interrupt register (*MInt*). When set, IBDisp will trap instead of dispatch (see sec. G.5).

MInt has two uses in the Mesa Emulator: to catch pending interrupts (given by uWP) and to indicate that the PC is pointing to the last word of a page (uPCCross true). Thus, *MInt* being set does not necessarily imply there are pending interrupts.

IO tasks should set *MInt* in the same click in which uWP is written. uWDC does not need to be checked before setting *MInt*. Similarly, the Emulator should reset *MInt* (ClrIntErr) in the same click in which uWP is read and reset.

R ← uWP, MesaIntRq,	c1;
R ← R or uWakeupMask,	c2;
uWP ← R,	c3;

2. **EnterKern:** The Kernel task will run in the next click when EnterKern is executed in c1. No other tasks can run when the kernel is running, so it becomes necessary for the Kernel to refresh memory. EnterKern is used to implement breakpoints.

3. **ExitKern:** When executed in c1, ExitKern will cause normal task scheduling to begin. Thus, which task runs in the click following ExitKern depends on where in the round structure the ExitKern occured.

4. **Refresh:** When *fY* or *fZ* = *Refresh* in c1, one row of all the memory chips is refreshed (a "RAS only" memory cycle). The row address is supplied by a 7 bit counter in the memory controller which is incremented once per Refresh. All 128 memory rows must be refreshed every 2 mSec, which is equivalent to about 1 refresh per 33 clicks (or twice per display line, 14 rounds).

If Refresh is executed in c2 or c3, or when the display is reading from the low bank (i.e., Refresh in clicks [0..3] and Display on & not blanking) no memory refresh occurs.

5. **Clear Wakeup Requests:** ClrDPRq, ClrIOPRq, and ClrRefRq reset the Display/Printer, IOP, or Refresh task wakeup requests. The Ethernet and Disk wakeups are cleared automatically by the hardware after being serviced. ClrKFlags clears the Disk Word Status register.

6. **Noop's:** *fX*, *fY (fYNorm)*, or *fZ (fZNorm)* = 8 are Noop's. *fZ* = 9, 0A, and 0B are also Noop's because these values are not decoded. *fZ* = 4 *(fZNorm)* is unused. *fY* = 0B and 0E *(IOOut)* are unused and available on the backplane.

N. MASS

MASS creates control-store-loadable files from microcode source files. It also produces symbol table files for Burdock and a status file with error messages which refer directly to the source text.

1. Files: There are two types of source files. There are macro-and-definitions files and there are microcode source files. The macro-and-definitions files have a .df extension and source files have an .mc extension. .df files do not contain microinstructions and the definitions are global to an assembly. Some intermediate files (.ml, .si. and .eb) are produced for each source file to allow subsequent partial re-assemblies.

Source files may also have macros and definitions (which will be used globally), however this may produce inconsistent results if the macros or definitions are used by another .mc file.

The .fb is the control-store-loadable file of allocated microinstructions. The .ft is a human-readable form of the .fb file. The symbol table file has an .st extension.

The .er status file contains MASS version information and the errors encountered. Along with the explanatory text and a copy of the source line in error, the phrase which caused the error is shown. There are usually multiple error messages per incorrect source line since MASS outputs a message for each possible and unsuccessful attempt to encode the statement. Allocation errors are displayed by listing, for the entire block, the labels and their constraints.

The following files are used (u) or generated (g) by MASS in pass 1 or pass 2, where pass 2 is the allocation pass.

	pass1	pass2	
.mc	u	--	MicroCode source
.df	u	--	macro and Defs File
.eb	g	u	Early Binary (one per .mc)
.ml	g	u	Label constraint records & Reserves (one per .mc)
.si	g	u	Symbol Intermediate (one per .mc)
.fb	--	ug	Final Binary
.ft	--	g	Final binary Text
.st	--	g	Symbol Table
.er	g	g	status and ERrors

2. Command Line Switches: Switches should only be appended to a file name, not to **MASS**. A typical command line might be

> MASS Dandelion/d Name Name/o/t/a

Local Switches:

none	use **Name** for .mc input, .eb, .ml, & .si output files.
/d	designates a .df file
/o	use **Name** for .er, .fb, .ft, and .st Output files
/2	pass 2 only for this file (will read .eb, .ml, & .si files)
/x	satisfy imports from this file but exclude its locations from allocation

Global Switches:

/a	Automatic re-assembly: For each .mc, pass1 executes only if there is no corresponding .si file or the .mc file has been updated since the last .si written.
/n	.ft and .fb not generated
/t	.ft not generated
/g	MASS doesn't pause with "hit me." Hits itself.

3. **Sample Command Lines:** The following three examples illustrate file creation and usage during an assembly.

MASS Regs/d Macros/d Source System/o

```
pass1:   reads Regs.df, Macros.df, Source.mc
         writes Source.eb, Source.ml, Source.si
pass2:   reads Source.ml, Source.si
         writes System.st
         allocates Source
         reads Source.eb
         writes System.fb, System.ft, System.er
```

MASS Regs/d SourceA SourceB System/o/t/a

```
pass1:   reads Regs.df, SourceA.mc
         reads SourceA.mc or SourceB.mc if they have been updated
pass2:   reads SourceA.ml, SourceA.si, SourceB.ml, SourceB.si
         writes System.st
         allocates Source, SourceB
         reads SourceA.eb, SourceB.eb
         writes System.fb, System.er
```

MASS Config/d SourceA/2 SourceB SourceC/x System/o

```
pass1:   reads Config.df, SourceB.mc
         writes SourceB.ml, SourceB.eb, SourceB.si
pass2:   reads SourceA.ml, SourceA.si, SourceB.ml, SourceB.si, SourceC.fb
         writes System.st
         allocates SourceA, SourceB
         reads SourceA.eb, SourceB.eb
         writes System.fb, System.ft, System.er
```

4. **MASS Runtime Interface.** While MASS is running, the cursor is moved vertically down for each .mc or .df file read, and horizontally for each error encountered. The cursor is reversed once per statement processed.

During the allocation phases (pass 2), a square cursor is displayed with the number of enclosed dots increasing to six. If errors are found in the assembly, MASS pauses with the cursor showing "hit me". As soon as any keyboard character is struck, MASS returns to the Alto executive (which may continue on to Bravo if the S macro was previously invoked). The Swat key causes MASS to close the .er file, but might produce an invalid .si file, so the next assembly should not use the /a switch.

5. **Microcode Source Format:**

(a) **Comments:** All text between squiggly bracket pairs { and } is ignored by MASS. The brackets nest, so they must be properly paired. This feature enables commenting out sections of code which are already commented.

(b) **Names & Numbers:** Names are used as instruction labels (Label:), constants (Set), register names (RegDef), macro names (MacroDef), or names of microinstruction functions (reserved macro names). Names do not start with digits and can not have imbedded spaces. The case of letters in names must be consistent.

None of the function field or builtin macro names should be used as user names (i.e., as labels, constants, registers, or macros). For example, RH, ib, Q, EStatus, c1, xor, and pop are all reserved. Null and Apass are also reserved names. ← should not be present in a user name.

Hexadecimal numbers are the default or can be specified with a trailing 'x. A decimal number is specified by a trailing 'd and an octal number by a trailing 'b. Hex numbers which begin with A..F must be prefixed by a zero. For example, 15'd = 17'b = 0F'x = 0F.

(c) Source Line Format: A line of source which defines a single microinstruction is a list of clauses which is terminated by a semicolon. Clauses are separated from one another by commas. A clause is either a microinstruction function name, a macro invocation, or an arrow (arithmetic) clause.

A name followed by a colon is the label of the microinstruction. Microinstructions can have multiple labels.

Spaces are used as token delimiters in arrow clauses, but are otherwise ignored. Parentheses may be used in arithmetic clauses where they may improve readability and are ignored by MASS. Parentheses are also used in macro definitions. Brackets are used to denote the argument list to macros, but are also used in MAR←[...], Map←[...], and []←.

(d) Register & Constant Definitions: All registers and constants must be defined before their names are used. The builtin macro RegDef is used to associate a name with a register type {R, RH, U, UY} and value. An error message will be given if a register name is used twice.

```
RegDef[Reg, R, 0B];
RegDef[uReg, U, 4E];
RegDef[rhReg, RH, 4];
RegDef[uBlock, UY, 9];
```

The macro Set is used to associate a number with a name which can be used in an arithmetic clause (as a constant) or in other macros.

```
Set[sZeroDivisor, 0C];
Set[L0.ERefill, 8];
```

(e) Arrow Clause: Arrow clauses (which include arithmetic clauses) may have one or more destinations and there can be more than one arrow clause per source line.

```
B ← B - A,                           c1;
B ← uReg ← stackP ← B or A,          c2;
R ← MD, Ureg ← RA,                   c3;
```

The right side of an arrow clause consists of a single item, or two items with an operator, or three with two identical operators. The operators are: +, -, or, and, xor. It is also possible for an item to have a qualifying unary operator (~ or -). For example,

```
B ← ~cFieldSize,                     c1;
B ← A + B + 1,                       c2;
B ← ~0FF and B,                      c3;
```

(f) Default Labels: A $ can be used in BRANCH, DISPn, and CANCELBR macros and refers to the address of the following source statement.

```
ZeroBr, BRANCH[MapUpDate, $],        c1;
CANCELBR[$],                         c2;
```

Also, the label used by all the at clauses of a dispatch table need not actually be the label of any particular statement of the table. Such floating labels do not appear in the symbol table.

(g) Cycle Numbers: Each source line must include a cycle number macro: c1, c2, c3, or c*. MASS checks whether all instructions in cycle n are followed by instructions in cycle (n + 1 mod 3) or are preceeded by instructions in (n-1 mod 3).

The c* macro inhibits the wrong cycle error from MASS. This can be used in loops which are not a multiple of three instructions but always exit on the same cycle. It is also useful for subroutines which don't contain cycle constrained operations and therefore can be called from statements on different cycles or return to various cycles. Such loops or subroutines should not contain MAR←, Map←, MDR←, ←MD, AltUaddr, or any other cycle-dependent operations.

(h) User Macros: A user macro is defined by supplying a name for the macro and a text string which will replace the name. Macros return nothing or a single number or variable, but never a pair of arguments, for instance. Up to 9 arguments may be supplied in the macro call and are referenced in the expansion as #n for n = 0..9. The special argument #0 is replaced by the number of arguments in the macro call and #1..#9 are replaced with the appropriate argument (or Null if there is none). Parentheses must be used to enclose arguments containing commas. For example:

MacroDef[NewMacro, (OldMacroA[#1], OldMacroB[#2, #3])];

The invocation of a macro is caused by supplying the macro and optionally up to nine arguments. Arguments are separated by commas, and if an argument contains a comma, it must be enclosed within parentheses. Macro calls can not appear on the left side of arithmetic clauses, and only unargumented macros (such as macros defined by Set) can appear on the right side.

If an .mc file contains macro definitions, they remain valid for subsequent .mc files which are part of the same assembly unit.

(i) External Variables: Note: Since MASS assembles fairly rapidly, these macros for constructing large systems are almost never used.

The macro IMPORT indicates which labels of the .mc file have been defined elsewhere. The EXPORT macro specifies labels which other modules will import. When MASS is assembling a group of files together (a combination of .mc and/or .ml/.si files) it is illegal to have the same label defined twice or imported and defined. Labels defined in IMPORT macros will be assigned values from the EXPORT of a previous assembly unit by using the /x switch on the file name.

6. Starting Address: SetTask[n] and StartAddress[Label] place special entries in the .fb file which Burdock uses when loading the program. Specifically, Burdock initializes $TPC[n]$ to the value of Label. The task specified by the SetTask macro remains in effect until the next SetTask.

Note that the statement at Label must have a CANCELBR[$, OF], since the hardware TC (conditions) register can not be initialized. Since there are no explicit task-specific user registers in the Dandelion, the SetTask macro has no other effects.

7. Conditional Code Generation: The IfEqual, IfGreater, IfAndZero and SkipTo macros can be used to conditionally assemble sections of code. IfEqual, IfGreater, and IfAndZero have four arguments [x, y, resultA, resultB], where resultA and resultB are names (macros, registers, etc.). They are defined by:

```
IfEqual   ←  IF  x = y  THEN  resultA  ELSE  resultB
IfGreater ←  IF  x > y  THEN  resultA  ELSE  resultB
IfAndZero ←  IF  (x  and  y)  =  0  THEN  resultA  ELSE  resultB
```

SkipTo[Label] causes MASS to stop processing source lines until the line labeled Label!, where the ! must be appended to the label name. An example conditional assembly macro:

IfEqual[Config, 1, , SkipTo[BandBLTEOF]];

8. Command Line Set's: The values of variables can be set on the command line by phrases of the form [varName,value] with no spaces inside the brackets. MASS treats the bracketed pair as an argument to a Set macro. MASS predefines a variable, Config, to be zero which is useful for controlling system configurations.

9. Reserves: Reserve[LowLoc, HighLoc], where LowLoc and HighLoc are numbers, causes MASS not to allocate instructions within the range. Reserve[Loc] will cause MASS not to place an instruction at control address Loc.

10. Arithmetic Macros: The following macros return a value which is the operation applied between the (up to 9) arguments: Add, Mul, And, Or, and Xor. Lshift[arg1, arg2] returns the value of arg1 left shifted by arg2, Rshift[arg1, arg2] shifts right, and Sub[arg1, arg2] returns the value of arg1 - arg2.

11. Builtin Macro Summary: Most of the following macros have already been discussed.

Add[arg1, ... , arg9]
Mul[arg1, ... , arg9]
And[arg1, ... , arg9]
Or[arg1, ... , arg9]
Xor[arg1, ... , arg9]
Lshift[arg1, arg2]
Rshift[arg1, arg2]
Sub[arg1, arg2]

GOTO[label]
CALL[label]   {equivalent to GOTO}
GOTOABS[value]
BRANCH[label0, label1, mask]   {mask optional}
CANCELBR[label, mask]   {mask optional}
DISP2[label, mask]   {mask optional}
DISP3[label, mask]   {mask optional}
DISP4[label, mask]   {mask optional}
DISPNI[label]   {mask optional}
RET[label]   {equivalent to DISP4}
c1, c2, c3, c*

IfEqual[x, y, equalVal, unequalVal]
IfGreater[x, y, equalVal, unequalVal]
IfAndZero[x, y, equalVal, unequalVal]
SkipTo[label]

EXPORT[label1, ..., label9]
IMPORT[label1, ..., label9]
at[offset, modulo, label]   {modulo and label optional}
Reserve[LowLoc, HighLoc]   {HighAddr optional}
LOOPHOLE[type]

Set[varName, value]
MacroDef[macroName, expansion]
RegDef[regName, regType, regAddress]
PrintVar[varName]   {print variable in .er file}
Print[Message]   {puts Message in .er file (only letters, numbers, and ".")}
SetTask[task]
StartAddress[label]

11. LOOPHOLE[] Summary: Most of these loophole phrases have already been discussed.

wok	{previously WriteOK}
mdok	{allows ←MD without MAR←}
mdrok	{allows MDR← without MAR←}
stw	{previously SuppressTimingWarning. Use niblTiming & byteTiming instead}
niblTiming	{Use Nibble timing for this instruction}
byteTiming	{Use Byte timing for this instruction}
pci	{Suppress pageCross Inverted error message}
natc	{Prevents attributes of Label in at from causing allocation errors}

O. Microcode Conventions:

1. Font: All microcode source files should be in either Helvetica8 or Helvetica10.

2. Tabs: The tab stops for Helvetica8 microcode should be near 140, 160, and 420 points. The stop at 160 keeps lines with long labels from jumping over to the comment field when Bravo is not in hardcopy mode.

3. Source File Header: The top of the microcode source file should have at least the following information:

```
{File name: <Name>.mc
Description: <what file contains>,
 Author: <being>,
 Created: <date>,
 Last Edited: <date & time>, <what changed>}
```

4. Source Line Format: The general format of a source statement is:

```
<Label:>      <Arrow clause>, <functions>, <DispBr>, <Ln←,LnDisp>, <GOTOfield>      ,<cycle>, <at>;
```

There should always be at least one space after a comma and one before and after ← when part of an arithmetic clause.

The <cycle> and <at> clauses should be located after the 420 point tab. It also helps to locate the last comma at the 420 stop. Imbedded comments are encouraged. For example:

```
        TOS ← TOS + 1                                         ,c3;

Shift:  [] ← ~0F and TOS {mask out low part}                  ,c1;
        PC ← PC + PC16, L1 ← L1.where                         ,c2;
        TT ← STK {TT ← v}                                     ,c3;

        TOS ← T LRot1, DISP4[MaskTbl]                         ,c1;
```

5. Click Spacing: Statements which are executed in the same click should be preceded and followed by a blank line. This makes the microcode infinitely more readable.

6. Use of at's: MASS generates the required at phrases for BRANCH's and Ln←'s, but not for DISPn's or RET's. For branch destinations which are widely separated (by a page of text for instance), it helps if at macros explicitly identify the corresponding partner(s) of the pair or group.

```
MulLoop:  Ybus ← Q and 1, NZeroBr,                           c1, at[0,2,MLDEnd];
          TT ← TT - 1, ZeroBr, BRANCH[MPlier0, Mplier1, 0E], c2;

MPlier0:  T ← DARShift1 (T + 0), BRANCH[MulLoop, MLDEnd],     c3;
MPlier1:  T ← DARShift1 (T + TOS), BRANCH[MulLoop, MLDEnd],   c3;

MLDEnd:   STK ← T {long.high/rem}, pop {point at s},          c1, at[1,2,MulLoop];
```

7. Names: If names consist of multiple parts, the first letter of each sub-name should be capitalized. All names should not be all capitalized: MulLoop, FrameOffset, MapOK. The letter i is better to use than the capital form I (I), which is indistinguishable from l (small L) and similar to 1 (one). Try to avoid a solitary O in a name since it looks like a 0 (zero).

All instances of all names must be identical with respect to capitalization, else MASS will not recognize them. Bravo searching is also more productive.

U register names should start with the letter u or U. RH register names should begin with rh or RH and otherwise equal the corresponding R register name (unless they are totally unrelated). It is also useful to start constants with a c or C and R registers with an r. For example: uTemp, uXferCmd, rhEE, cHeadMask, rRcvCnt.

8. U Register Constraints: When the address of a U register must be constrained by the *rA* or *fZ* field, these restrictions should be commented.

```
RegDef[UreturnPC, U, 5D];  {rA = PC, fZ = ←ib}
RegDef[UvC, U, 2B];  {fZ = ←RH};
```

9. Link Register Constants: Constants defined by Set which will be used in Ln← macros, should be prefixed by Ln., for example L3 ← L3.Catch. Also, any constraints on the link register value should be commented.

```
Set[L0.rfR0, 4];  {must be even}
Set[L1.WstrR, 0D];  {must end in 01}
```

10. Register Definitions & IO Page: The allocation and definition of registers used in the standard microcode system are given in [Iris]<Workstation>Dandelion.df. It also includes the definition of the IO page.

Dandelion.df should ONLY be written via the Librarian Access Tool (It can be read anytime by FTP). Access checkOutReason/r Dandelion/e is used to check Dandelion.df out and move it to your disk, and Access Dandelion/s is used to return it to Iris.

11. Bravo S macro: The S macro facilitates ping-ponging between Bravo and MASS. It assumes standard file naming conventions. In particular, corresponding to a Name.mc source file, there is a Name.cm command file, and errors are placed into Name.er. [Iris]<Workstation>MASS>user.cmslice contains the macros.

P. Burdock & the CP

Burdock is a client of the Tools package, thereby making available to the user multiple windows and other tools, such as source windows and the file tool. There are four special windows which are always present in Burdock: CP Panel, IOP Panel, State Analyzer, and Files. This section deals with the CP Panel and the State Analyzer as connected to the CP.

1. CP Panel Commands: This section discusses the following CP Panel commands: *Boot, Load, Start, Stop, Reset,* and *Continue.* Other menu commands (*LoadReal, Resume, UnBreak*, ListBreaks, Alternate, ClrPanel, ClrBreaks,* & *Update*) are not covered here.

The *Boot* command boots the IOP if necessary, and then loads the CP kernel (Kernel.fb). All the *TPC's* are initialized to 0FEF, which currently is a microinstruction which loops on itself and resets the display controller. Note that while the kernel is executing at the kernel task level, no other tasks can run.

Load places the .fb file into the control store and reads the .st symbol table file into Burdock. The file name comes from the Files window. If the source files contained SetTask and StartAddress macros, the specified *TPC's* are initialized.

Start writes the Emulator (task 0) *TPC* with the value of the type-in label and then causes the CP to exit the kernel. Normal task scheduling begins, and if multiple tasks are enabled, any one of them can begin executing first. All *TPCs* must be properly set (see *Reset*). Also, the first instruction to be executed at each task level must have been assembled with a CANCELBR[$, 0F] macro.

Stop causes the kernel task to run, thereby blocking all other tasks. The tasks which were running are interrupted on a click boundary, so their *TPC's* and *TC's* remain valid. It is not possible to determine which click boundary in a round the stop occured on.

The *Reset* command reinitializes all the *TPCs* to their values given by the SetTask/StartAddress macros of the source files.

Continue exits the kernel given the current values of the *TPCs* and pending conditions (*TCs*). Like *Start*, it is not known which task will begin executing first.

2. Breakpoints: *Break* sets a breakpoint at the address given by the type-in. Burdock saves the breakpointed microinstruction in an internal table and replaces it with the appropriate instruction which will cause entry into the kernel. If the control store is examined from Burdock, the breakpoints will be visible instead of the original instructions.

Up to 16 microinstructions can be simultaneously breakpointed. Each is assigned a breakID in [0..0F], which Burdock uses to identify the breakpoint. (The breakID can be found in the *fZ* field of the breakpointed location.)

However, with respect to *Continue*ing from a breakpoint, c2 and c3 breakpointed instructions are treated differently from c1's. In particular, pending branch/dispatch bits or memory data are lost. Generally, in order to guarantee correct restart from c2 or c3 breakpointed instructions, they should not be preceeded by a branch or dispatch phrase or be in clicks beginning with a MAR←. However, it is possible to specify the pending dispatch bits by writing them into the U register UKSaveDisp before *Continue*ing.

This affliction does not affect c1 breakpoints, and was only applied to c2 and c3 breakpoints due to control store economy considerations. (It is possible to write a Kernel which saves c2 & c3 pending bits and also saves memory data and subsequently restarts the memory on continuation.)

If a cycle1 breakpoint is executed, the kernel will always run in the following task. However, if a c2 or c3 breakpoint is executed, the breakpointed task must run for at least one more click in order for the breakpoint to take effect. This is never a problem with the emulator (since it will eventually run), but could be a problem with an IO task if the task's request were disabled during the breakpointed click, thereby preventing it from executing again. Note that for c2 and c3 breakpoints it is not known which other tasks may have run between the breakpointed click and the kernel entry.

If *Start* is used instead of *Continue* after a breakpoint, the second instruction executed must be at[0F,10]. This requirement is not necessary if UKSaveDisp (or *TC[task]* in the case of a *Stop*) does not have any non-zero bits where the low 4 bits of the address of the second instruction has zero bits. For example, if UKSaveDisp or *TC[task]* is 0, then any microinstruction can be *Start*'d. If UKSaveDisp or *TC[task]* is 1, then only microinstructions which are followed by an instruction at an odd address can be started.

Another feature of breakpointing (or *Stop*ing) is that on entry to the kernel the display controller is set such that it will not read from the low bank (DCtl ← 3, On & Blank). (It can't be turned off since it may have been on. It's not possible to read this On/Off state bit). This allows the kernel to do Refreshs and a user to read the low bank of memory from Burdock. Note that the display will be left in this state on exit from the kernel.

UnBreak restores the breakpointed location with its original contents and makes the breakID available again.

3. CP Panel Registers: After the CP has been *Boot*'d or *Stop*'d, its registers can be read or written via absolute addresses. After a program has been loaded, Burdock can read or write registers given their source file symbolic names. Burdock ignores capitalization in all names (e.g., rCnt and RCnt are equivalent to Burdock). Note that the control store, *TPC*, and *TC* registers can be read or written if only the IOP is booted (& not the CP).

The format for reading and writing a register with absolute addresses is .name address or address .name. U, R, and RH registers can be read or written with their symbolic name. A number without a .name is assumed to be a virtual memory address. All numbers in Burdock are hexadecimal. Burdock recognizes the following CP register names:

.name	address range	
.q		
.r	[0..0F]	
.rh	[0..0F]	
.u	[0..0FF]	
.tc	[0..7]	{read only}
.tpc	[0..7]	
.link	[0..7]	
.ioxin	[0..0F]	{address is value of *fZ*. Read only}
.ioout	[0..0F]	{address is value of *fY*. Write only}
.cr	[0..0FFF]	{control store: takes 3 data words}
.mr	[0..3FFFF]	{memory real address}
.mv	[0..3FFFFF]	{memory virtual address: .mv optional}
.map	[0..3FFF]	{address is index into Map}
.ib		
.ibPtr		{can only be set to *word* or *byte*}
.stackP		
.pc16		{set to 0 or 1}
.MInt		{set to 0 or 1}
.EKErr		{read only}

4. State Analyzer Window: For CP debugging, the low 12 bits of the state analyzer (Tektronics 7904 + DF1 formatter) are connected to the control store address lines *NIA[0..11]*. The upper 4 bits are available for other inputs (such as *IOPWait*, cycle number, task number, etc). The Analyzer window displays 256 addresses either before, after, or in the middle of a trigger address entered into the Tektronix hardware. (Burdock can translate between absolute control store address and symbolic labels.)

The window can format, filter, and search the addresses for matches. The "mask" is and'd with all displayed addresses and the "xor" mask is used to invert appropriate bits. The default for the xor mask is 0FFF since the control store address lines are inverted.

The addresses can be displayed in one of 4 modes: NIA (i.e., symbolic labels), hex, octal, or binary. Since *NIA* is displayed, the occurrence of a label corresponds to the cycle when the instruction was being read from the control store; the indicated instruction was actually executed in the following cycle. Thus, if c1 is being displayed in the top 4 bits, it actually corresponds to microinstructions labeled c2 in the source file.

Q. Timing Constraints

The architecture of the CP allows the execution of microinstructions which will not always properly complete. This is due to either "slow" X bus operands or "slow" destination registers, i.e., hardware timing differences between registers and/or operations. These timing variations exist mostly on the X bus. MASS will flag such instructions with a timing violation error.

1. Non-Xbus Operations: All ALU internal register-to-register operations complete on time. All Y-bus destinations can be loaded as a result of any ALU operation which does not use the X bus as an operand (except for the high 12 bits of SU register arithmetic).

2. XBus Operations: If the ALU operation uses the X bus as an operand ($aS = D,A, D,Q, D,0$), it may not complete within 137 nanoseconds on some machines under some circumstances. In general, all X bus sources can at least be loaded into an R register, which is a logical operation ($aS = D,0, aF = RorS$).

Figure 7 should answer the question: "Is my microinstruction legal with respect to X-bus timing?" The table deals with all possible X-bus sources and destinations: X-bus-source-to-X-bus destination, X-bus ALU operands ($aS = D,A, D,Q, D,0$), and X-bus branching and dispatching. Intersections marked with a square indicate legal source/destination combinations or branching phrases.

X + R represents the 3 arithmetic operations ($aF = R+S, S-R, R-S$) and X or R the 5 logical operations ($aF = RorS, RandS, \sim RandS, RxorS, \sim RxorS$). B← implies the loading of an R register; Q← has the same timing. pgCross refers to the automatic page cross branch with MAR← and pageCross & OVR refer to PgCrOvDisp.

Branching and dispatching have different timing than the basic ALU operations and a potential statement must meet both conditions. In general, zero, negative, or overflow branching is not possible with any X-bus operand.

3. ALU Arithmetic: The ALU performs arithmetic at three different speeds depending on which bits of the result you're looking at. Thus, figure 7 has three numbers for arithmetic operations depending on which bits of the result are of interest. ALU[0-7] are the slowest since they depend on a carry from the lookahead unit. ALU[8-11] are next as they depend on a ripple carry from the low nibble. Finally, ALU[12-15] are fastest since Cin arrivies very early relative to X bus sources. Thus, the low nibble always has the timing of a corresponding ALU logic operation.

Note that some "+1" or "-1" operations do not necessarily imply use of the X bus, but use Cin instead. Thus, R ← R + 1, NegBr is legal where R ← R + 2, NegBr is not.

All arithmetic operations with the ALU internal zero as an operand ($aS = 0,Q, 0,B, 0,A$, or $D,0$) complete on time. This obviously includes all X-bus sources.

4. Timing LOOPHOLEs: MASS checks for the timing violations given in Figure 7. The macros LOOPHOLE[niblTiming] or LOOPHOLE[byteTiming] can be used to prevent an error message where the timing is legal for the bottom 4 or 8 bits. LOOPHOLE[stw] can be used in other cases.

```
R ← Ureg - 1, LOOPHOLE[byteTiming]          c1;
MDR ← R + 2, LOOPHOLE[niblTiming]           c2;
```

X Operation	X setup	X Source							
		SU	MD	RH	constant ←ib ErrIBStkp	IOIn	A LRotn	(A or B) LRotn	(A + B) LRotn
X Source Time		75	97	74	59	63	91	102	131 127 105
B←X or R	40	■	■	■	■	■	■	—	—
B←X or R, ZeroBr	58	■		■	■	■		—	—
B←X or R, NegBr	58	■		■	■	■		—	—
[]←X or R, YDisp	68			■	■	■		—	—
B←LShiftl (X or R)	50	■		■	■	■	—	—	—
B←LRotl X (X or R)	60	■		■	■	■	—	—	—
MAR← X or R	78		—		■		—	—	—
Map← X or R	78		—		■		—	—	—
MDR← X or R	45	■	—	■	■	■	—	—	—
SU← X or R	87	—						—	—
IOYOut← X or R	64	■		■	■	■	—	—	—
B←X + R	74 65 40	◢	▸	■	■	■	▸	—	—
B←X + R, ZeroBr	95							—	—
B←X + R, NegBr	87							—	—
B←X + R, OVR	90							—	—
B←X + R, CarryBr	80				■			—	—
B←X + R, NibCarryBr	58	■		■	■	■		—	—
B←X + R, PgCarryBr	65	■		■	■	■		—	—
B←X + R, pageCross	77				■	■		—	—
MAR←X + R, pgCross	72			■	■	■			
B←X + R, YDisp	68			■	■	■		—	—
B←RShiftl (X+R)	89 80 50	▸		▸	◢	▸	—	—	—
B←RRotl (X+R)	99 90 60	▸		▸	▸	▸	—	—	—
MAR← X + R	78		—		■		—	—	—
Map← X + R	110		—				—	—	—
MDR← X + R	77 70 45		—	■	■	■	—	—	—
SU← X + R	119 112 87	—						—	—
IOYOut←X + R, stackP←	80 73 48	▸		■	■	◢	—	—	—
Xbus ← X, XDisp	32	■	■	■	■	■	■	■	
RH ← X	36	■	■	■	■	■	■	■	
IB ← X	37	■	■	■	■	■	■	■	
IOYOut ← X	22	■	■	■	■	■	■	■	▸

■ Timing OK across 16 bits of result

◢ Timing OK across low byte of result

▸ Timing OK across low nibble of result

— Operation not possible due to syntax or data paths

Figure 7. Allowable X-bus Operations

2 Dec 80

Appendix: Antithetical List of MicroInstructions

This appendix contains a list of some example microinstructions in addition to examples of illegal ones. They are clearly not in the correct MASS format. This is not a complete list!

Those microcode statements which can be written but don't work as intended have three possible reasons for their shortcomings:

1. Timing error,
2. Syntax error, or
3. Characteristic of a processor data path.

Abbreviations used here:

SU :: = STK | U register;
A :: = the R register addressed by rA;
B :: = the R register addressed by rB;
R :: = an R register addressed by either rA or rB;
Rot1 :: = LRot1 | RRot1;
Shift1 :: = LShift1 | RShift1;
LRotn :: = LRot0 | LRot4 | LRot8 | LRot12;
constant :: = Nibble | Byte;
ArithBr :: = NegBr | ZeroBr | PgCrOvDisp | CarryBr | PgCarryBr;
LogicBr :: = NegBr | ZeroBr;
XDispBr :: = XRefBr | XHDisp | XwdDisp | XLDisp | XDisp;
° :: = alu logic operation (R or S, R and S, ~R and S, R xor S, ~R xor S);
± :: = alu arithmetic operation (R + S, S - R, R - S);
⊕ :: = alu arithmetic or logic operation;

1. General:

Possible: *Not Possible:*

B ← R ° R,
B ← R ± R,

Q ← R ⊕ R,	B ← Q ← R ⊕ R,	(2)
YBus ← A, B ← R ⊕ R,	YBus ← A, [] ← R ⊕ R,	(2)
	YBus ← A, Q ← R ⊕ R,	(2)

2. SU registers:

Possible: *Not Possible:*

B ← SU ° A,	B ← SU ± A,	(1)
B ← SU ° Q,		
SU ← R ° R,	SU ← R ± R,	(1)
B ← SU ← R ° R,		
SU ← A, B ← R ° R,		
SU ← A, B ← R + R + 1,	SU ← A, B ← R + R, Cin←0,	(2)
SU ← A, B ← R - R,	SU ← A, B ← R - R - 1,	(2)
	SU ← A, B ← SU,	(2) & (3)
	SU ← SU ⊕ A,	(2) & (3)

SU ← A, B ← R ± R ± PC16,

3. Constants:

Possible:

B ← -constant,
B ← A ⊕ constant,
B ← 0,
{B ← 100,}   B ← 0FF + 1
{B ← 0FF00,}   B ← ~0FF
{B ← 0FFFF,}   B ← ~B xor B
{B ← 7FFF,}   B ← RShift1 (~B xor B)
{B ← 1FF,}   B ← LShift1 0FF, SE←1

SU ← 0
{SU ← 0FFFF,}   SU ← ~A xor A

Not Possible:

STK ← constant,   (1)
U ← constant,   (2)
B ← SU ⊕ constant,   (2) & (3)

4. RH registers:

Possible:

B ← RH[B] ° A,
B ← RH[B] ± A,
RH[B] ← SU,
RH[B] ← SU, B ← SU ° A,
RH[B] ← SU, B ← R ⊕ R,
RH[B] ← constant,
RH[B] ← constant, B ← R ⊕ R,
RH[B] ← ib,
RH[B] ← ib, B ← R ⊕ R,
STK ← A, B ← RH[B] ⊕ A,

Not Possible:

A ← RH[B],   (2)
RH[B] ← RH[B]

U ← A, B ← RH[B] ⊕ A,   (2)

5. Memory:

Possible:

R ← MD,
R ← R ° MD,
Q ← R ° MD,
RH[B] ← MD,
RH[B] ← MD, B ← R ⊕ R,
RH[B] ← MD, B ← MD, SU ← A,
MAR ← [rh[B], A], B ← B ⊕ R,
MAR ← [rh[B], A], B ← IOIn,
MAR ← [rh[B], A], B ← RH[B],
MAR ← [rh[B], A], B ← SU,
MAR ← [rh[B], A], B ← constant,
MAR ← [rh[B], B ⊕ R],
MAR ← B ← [rh[B], B ⊕ R],
MAR ← [rh[B], constant⊕ R],
MAR ← [rh[B], ib⊕ R],
Map ← Q ← [rh[B], R ⊕ R],
Map ← [rh[B], constant° R],
Map ← [rh[B], ib° R],
MDR ← R ⊕ R,
MDR ← A, B ← R ⊕ R,
MDR ← SU ° A,
MDR ← RH[B],

Not Possible:

R ← R ± MD,   (1)
R ← MD Shift,   (1)
SU ← MD,   (1)

MAR ← [rh[B], A], rh[B] ← x   (3)

MAR ← [rh[B], SU⊕ R],   (1)
MAR ← [rh[B], RH⊕ R],   (1)

Map ← [rh[B], SU⊕ R],   (1)
Map ← [rh[B], RH⊕ R],   (1)
Map ← [rh[B], constant± R],   (1)
Map ← [rh[B], ib± R],   (1)

MDR ← SU ⊕ A, {OK in bits[12-15]} (1)

6. IOIn/IOOut:

Possible:

B ← IOIn ⊕ A,
RH[B] ← IOIn,
MDR ← IOIn,
RH[B] ← IOIn, B ← IOIn ⊕ A,

IOOut ← (R ° R) LRotn,
IOOut ← Q LRotn,
IOOut ← MD,
IOOut ← IOIn,
IOOut ← RH[B],
IOOut ← SU,
IOOut ← A LRotn,
IOOut ← ib,

Not Possible:

SU ← (IOIn ⊕ A) LRotn,	(1)
IOOut ← (R ± R) LRotn,	(1)

7. stackP:

Possible:

stackP ← R ⊕ R,
stackP ← stackP ⊕ A,
stackP ← Nibble,
stackP ← IOIn,
stackP ← RH[B],
RH[B] ← stackP,

Not Possible:

stackP ← stackP + constant,	(2) & (3)
stackP ← MD,	(1)

8. LRotn:

Possible:

B ← A LRotn, {A bypass}
[] ← (A ° B) LRotn,
[] ← Q LRotn,
B ← A ° (A LRotn),

RH[B] ← A LRotn, B ← R ⊕ R,
RH[B] ← (R ° R) LRotn,

STK ← A, RH[B] ← A LRotn, B ← R + R + 1,
STK ← A, RH[B] ← (R ° R) LRotn,

Not Possible:

B ← (R ⊕ R) LRotn,	(2) & (3)
Q ← A LRotn,	(2) & (3)
B ← Q LRotn,	(2) & (3)
B ← B ⊕ (A LRotn),	(2)
SU ← (R ⊕ R) LRotn,	(2) & (3)
SU ← A LRotn,	(2) & (3)
MDR ← A LRotn,	(2) & (3)
B ← SU LRotn,	(2) & (3)
B ← (A LRotn) Rot1,	(2) & (3)
B ← (R ⊕ R LRotn) Shift1,	(2) & (3)
RH[B] ← A LRotn,	
RH[B] ← (R ± R) LRotn, {OK in [12-15]} (1)	
RH[B] ← (RH[B] ⊕ A) LRotn,	(2) & (3)
U ← A, RH[B] ← (R ° R) LRotn,	(2)

9. Single Bit Shifting:

Possible:

B ← (R ∘ R) Shift1,
B ← (R ∘ R) Rot1,

B ← (R + R) Shift1, SE←0,
B ← (R - R - 1) Shift1, SE←0,
B ← (SU ∘ A) Shift1, SE←0,

B ← (constant ± A) Shift1, {only [8-15]}
B ← R ± R, DRShift1,
B ← R ± R. DLShift1,

Not Possible:

Q ← (R ∘ R) Shift1,	(2)
Q ← (R ∘ R) Rot1,	(2)
B ← (R ± R) Shift1,	(1)
B ← (R ± R) Rot1,	(1)
YBus ← A, B ← R Shift1,	(2)
[] ← R Shift1,	(2)
B ← (R + R + 1) Shift1, SE←0	(2)
B ← (R - R) Shift1, SE←0	(2)
B ← (SU ∘ A) Shift1, SE←1,	(2)
SU ← B Shift1,	(2)
SU ← A, B ← R Shift,	(2)
B ← (constant ± A) Rot1 {OK in [12-15]}	

10. Branching:

Possible:

B ← R ⊕ R, Branch,
YBus ← A, B ← R ⊕ R, Branch,
B ← Xbus ∘ A, LogicBr,
B ← Xbus ± A, CarryBr, {except for SU, MD}
B ← Xbus ± A, PgCarryBr, {except for SU, MD}
B ← R ± 1, ArithBr,
B ← R ⊕ R, YDisp,
B ← RH[B] ∘ A. YDisp,

YBus ← A, B ← R ⊕ R, YDisp,
[] ← SU, XDispBr,
[] ← IOIn, XDispBr,
[] ← constant, XDispBr,
[] ← ib, XDispBr,
[] ← RH[B], XDispBr,
[] ← stackP, XDispBr,
[] ← MD, XDispBr,
B ← A LRotn, XDispBr,
B ← A ∘ (A LRotn), XDispBr,
[] ← (R ∘ R) LRotn, XDispBr,

Not Possible:

B ← MD ∘ A, LogicBr,	(1)
B ← Xbus ± A, ZeroBr,	(1)
B ← Xbus ± A, NegBr,	(1)
B ← SU ∘ A, YDisp,	(1)
B ← MD ∘ A, YDisp,	(1)

[] ← (R ± R) LRotn, XDispBr, {bits[12 = -15 OK} (1)